VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

E-learningová podpora výuky obecné rezoluční metody

E-learning Support for General Resolution Method

2011

Radek Daníšek

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava*.

V Ostravě 2. května 2011 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 2. května 2011 .....

Rád bych poděkoval panu Mgr. Marku Menšíkovi, Ph.D. za čas, informace, které mi poskytnul, ochotu a pomoc při řešení zadaného problému.

Abstrakt

Obsahem diplomové práce je vytvoření e-learningového systému, který studenty interaktivně vyučuje problematice obecné rezoluční metody a ověřuje také jejich znalosti. Systém kontroluje každý krok studenta a na základě požadavku je schopen i napovídat, jaký další krok má student udělat. Mimo vytvořený systém tato diplomová práce obsahuje také souhrn teorie predikátové logiky prvního řádu (PL1) související s touto problematikou a souhrn syntaktických důkazových metod v predikátové logice prvního řádu.

Klíčová slova: distributivní zákony, existenční uzávěr, extrakce literálů, formule, funkce, gramatika, instnace, klausule, konstanta, kontrola, kvantifikátor, logická spojka, nápověda, negace, PL1, proměnná, rezoluce, řetězcová struktura formule, speciální znaky, skolemizace, stromová struktura formule, syntaktická analýza, třída, unifikace, vnitřní znaky.

Abstract

Content of this diploma thesis includes the description of newly generated e-learning system, which interactively teaches students how to use general resolution method and also checks their knowledges. System checks each student´s step and is always able to help with next step in procedure of general resolution method. Except of newly generated system, the diploma thisis contains summary of Theory of first order predicate logic and summary of syntactic proof methods in first order predicate logic.

Keywords: check, class, clause, constant, distributive laws, existential closure, formula, function, grammar, help, instance, internal characters, literal extraction, logical connective, negation, parsing, quantifier, skolemization, special characters, string structure of the formula, tree structure of the formula, unification, variable.

Seznam použitých zkratek a symbolů

PL1	–	Predikátová logika prvního řádu
LL1	–	Gramatika dle levého rozkladu, analýzy zleva doprava, rozhodující se dle jednoho symbolu

Obsah

Seznam obrázků

1 Úvod

Jako téma pro diplomovou práci jsem si vybral tvorbu E-learningové podpory pro výuku obecné rezoluční metody. Jelikož je systém již nějakou dobu ve vývoji, mým úkolem je tento systém obohatit o modul, který bude zmíněnou problematiku obecné rezoluční metody v sobě zahrnovat.

Obecná rezoluční metoda se používá pro parciální automatické dokazování nesplnitelnosti formule. Pro předloženou nesplnitelnou formuli A metoda v konečném čase skutečnost nesplnitelnosti formule zjistí a zastaví se. Naopak v případě, kdy je formule A splnitelná algoritmus metody nemusí nikdy skončit.

Modul bude sloužit k automatizovanému dokazování logické pravdivosti formule, a proto se formule A bude vždy nejprve negovat a rezoluční metoda bude pracovat s negovanou formulí, aby byl algoritmus důkazu logické pravdivosti formule A konečný pro všechny logicky pravdivé formule. Pro logicky nepravdivé formule může být algoritmus modulu nekonečný. Nekonečnost algoritmu rezoluce se v řešení musí zakázat. Algoritmus rezoluce bude mít v řešení rekurzivní charakter. Problém potenciální nekonečnosti algoritmu bude v programu řešen pomoci pevně definované hranice rekurzivního volání (programu sebe sama).

Modul bude provázet uživatele také procesem převodu formule do klausulární formy, protože rezoluční metoda pracuje pouze s formulemi v klausulární formě.

Realizaci modulu je věnována kapitola 3. Pro vytvoření výše specifikovaného systému jsou nutné teoretické základy, jejichž souhrn je uveden v kapitole 2. Součástí této práce je i zhodnocení a sumarizace výsledného systému (modulu) uvedené v kapitole 6.

2 Souhrn teorie predikátové logiky prvního řádu (PL1)

V této kapitole se si vytvoříte vlastní představu o predikátové logice prvního řádu, pochopíte její základy a dokážete tyto základy sami používat například při řešení úloh a podobně.

Stěžejní literaturou pro kapitolu 2 byla skripta Matematická logika a přednášky paní Doc. RNDr. Marie Duží, CSc. k předmětu Matematická logika ([1], [3]).

2.1 Sémantický výklad predikátové logiky

Predikátová logika prvních řádu formalizuje úsudky o vlastnostech předmětů a vztazích mezi předměty pevně dané předmětné oblasti (univerza). Nezabývá se formalizací úsudků, které navíc vypovídají i o vlastnostech vlastností a vztahů a o vztazích mezi vlastnostmi a vztahy. Tím se zabývají **predikátové logiky druhého a vyšších řádů**. Predikátová logika 1. řádu je zobecněním výrokové logiky, kterou můžeme považovat za logikunultého řádu. Predikátová logika 1. řádu je postačující pro formalizaci mnohých matematických i jiných teorií.

Nyní si zavedeme definici jazyka predikátové logiky.

2.1.1 Jazyk predikátové logiky 2.1.1

1. ***Abeceda predikátové logiky*** je tvořena následujícími skupinami symbolů:
   (a) Logické symboly
      i. předmětové (individuové) proměnné: x, y, z, ... (příp. s indexy)
      ii. symboly pro spojky:$\neg$, $\wedge$, $\vee$, $\supset$, $\equiv$
      iii. symboly pro kvantifikátory: $\exists$,$\forall$
      iv. případně binární predikátový symbol = (predikátová logika s rovností)
   (b) Speciální symboly (určují specifiku jazyka)
      i. predikátové symboly: p,q,r, ... (případně s indexy)
      ii. funkční symboly: f,g,h, ... (případně s indexy) Ke každému funkčnímu a predikátovému symbolu je přiřazeno nezáporné číslo n (n $\geq$ 0), tzv. **arita**, udávající počet individuových proměnných, které jsou argumenty funkce nebopredikátu.
   (c) Pomocné symboly závorky: (,) případně i [,],{,}
2. ***Gramatika***, která udává, jak tvořit
   (a) Termy
      i. každý symbol proměnné je term
      ii. jsou-li $t_1, ..., t_n$ (n $\geq$ 0) termy a je-li f n-ární funkční symbol, pak výraz f($t_1, ..., t_n$) je term; pro n=0 se jedná o nulární funkční symbol, neboli individuovou konstantu (značíme a, b, c, ...)

iii. jen výrazy dle i. a ii. jsou termy

(b) ***atomické formule:***

i. je-li p n-ární predikátový symbol a jsou-li $t_1, ..., t_n$ termy, pak výraz p($t_1, ..., t_n$) je atomická formule

ii. jsou-li t_1 a t_2, pak výraz $(t_1 = t_2)$ je atomická formule

(c) ***formule:***

i. každá atomická formule je formule

ii. je-li výraz A formule, pak ¬A je formule

iii. jsou-li výrazy A a B forumle, pak výrazy (A∧B),(A∨B),(A⊃B),(A≡B) jsou formule

iv. je-li x proměnná a A formule, pak výrazy ∀xA a ∃xA jsou formule

v. jen výrazy dle i.-iv jsou formule

Následující poznámky ještě navíc upozorňují na důležité věci, které z definice hned nemusí být každému jasné.

2.1.2 Poznámky k definici jazyka predikátové logiky

1. Jazyk predikátové logiky, jak byl vymezen výše, je jazyk logiky 1. řádu, pro niž je charakteristické to, že *jediný přpustný typ proměnných jsou individuové proměnné.* Pouze individuové proměnné lze vázat kvantifikátory.

2. Definice jazyka umožňuje formulaci speciálního jazyka (určité teorie) konkretění volbou prvků (predikátových a funkčních konstant) dle bodu I)b. definice. Pro takový konkrétní jazyk budou platit obecné principy logické a mimo to - v závislosti na specifických vlastnostech (interpretacích) těchto prvků - i principy mimologické, které zadá tvůrce tohoto speciálního jazyka pomocí speciálních axiómů (dané teorie). Je-li arita funkčního symbolu $n = 0$, pak se jedná o individuovou konstantu (značíme a,b, ...), která však není pravou (logickou) konstantou, neboť podléhá (jako každý funkční symbol) interpretaci.

3. Zápis formulí můžeme zjednodušit na základě následujících konvencí o vynechávání závorek:

   - Elementární formule a formuli nejvyššího řádu netřeba závorkovat (vnější závorky vynecháváme).
   - Závorky je možné vynechávat v souladu s následující prioritní stupnicí funktorů: (∀, ∃), ¬, ∧, ∨, ⊃, ≡. Každý funktor vlevo od vybraného funktoru váže silněji než vybraný funktor.
   - V případě, že o paritě vyhodnocení nerozhodnou ani závorky ani prioritní stupnice, vyhodnocujeme formuli zleva doprava.
   - Speciálně vzhledem k asociativitě konjunkce a disjunkce, netřeba při zápisu vícečlenných konjunkcí a disjunkcí užívat žádné závorky.

- Vedle závorek (,) lze užívat i závorky [,],{,}.

K poznámkám je to v tuto chvíli vše a nyní přejdeme k dalším definicím, které je pro zvládnutí celé teorie nutné uvézt.

2.1.3 Definice - Vázaný výskyt, vázaná proměnná, volná proměnná, uzavřenost formule ...

Výskyt proměnné x ve formuli A je vázaný, jestliže je součástí nějaké podformule ∀xB(x) nebo ∃xB(x) formule A.

Proměnná x je vázaná ve formuli A, má-li v A vázaný výskyt. Výskyt proměnné x ve formuli A, který není vázaný, nazýváme volný.

Proměnná x je volná ve formuli A, má-li v A volný výskyt.

Formule, v níž každá proměnná má buď všechny výskyty volné nebo všechny výskyty vázané, se nazývá formulí s čistými proměnnými.

Formule se nazývá uzavřenou, neobsahuje-li žádnou volnou proměnnou. Formule, která obsahuje aspoň jednu volnou proměnnou se nazývá otevřenou.

Nechť $x_1, x_2, ..., x_n$ jsou všechny volné proměnné formule A. Potom uzvařenou formuli

$\forall A = {}_df \forall x_1 \forall x_2 ... \forall x_n A$ resp $\exists A = {}_df \exists x_1 \exists x_2 ... \exists x_n A$

nazýváme generálním resp. existenčním uzávěrem formule A.

Symbolem A(x/t) označujeme formuli, která vznikne z formule A korektní substitucí termu t za proměnnou x. Má-li být substituce korektní musí splňovat následujícídvě pravidla:

- Substituovat lze pouze za volné výskyty proměnné x ve formuli A a při substituci nahrazujeme všechny volné výskyty proměnné x ve formuli A.
- Žádná individuová proměnná vystupující v termu t se po provedení substituce x/t nesmí stát ve formuli A vázanou (v takovém případě je term t za proměnnou x ve formuli A nesubstituovatelný).

Symbolem $A(x_1, x_2, ..., x_n/t_1, t_2, ...t_n)$ označujeme formuli, která vznikne z formule A korektními substitucemi x_i/t_i pro i=1,2,..n.

Všechny formule tvaru $A(x_1, x_2, ..., x_n/t_1, t_2, ...t_n)$ nazýváme instanccemi formule A.

V tuto chvíli již máme určitou představu o jazyku predikátové logiky prvního řádu a o určitých jeho vnitřních pravidlech. Přejdeme k tomu, jak formule jazyka predikátové logiky chápat, tedy k sémantice jazyka (významu).

2.1.4 Sémantika PL1 - Interpretace formulí

Sémantika, neboli význam formulí predikátové logiky 1. řádu, je dána jejich interpretací. Máme-li danou formuli v PL1 a otázku na tuto formuli ohledně její pravdivosti, pak tato otázka ztrácí smysl, pokud nevíme, co formule znamená, tedy jak je interpretována.

Uvedu příklad: $\forall x P(f(x), x)$. Tento příklad může znamenat: pro všechna čísla platí, že jejich druhá mocnina je menší než ony samotné. Může však znamenat něco úplně jiného.

Nejprve tedy musíme stanovit, „o čem mluvíme“, tedy jaká je předmětná oblast - obor proměnnosti (individuových) proměnných, tj. zvolíme určitou neprázdnou množinu - universum diskursu, jijíž prvky jsou individua. Predikátové symboly mají vyjadřovat vztahy mezi těmito předměty. Přiřadíme tedy každému n-árnímu predikátovému symbolu jistou n-ární relaci nad universem. Jedná-li se o unární predikátový symbol, přiřadíme mu podmnožinu universa. Podobně funkčním symbolům přiřadíme n-ární funkce nad universem.

Teprve poté, co je formule interpretována, můžeme vyhodnotit její pravdivost či nepravdivost v zamýšlené interpretaci. Dalším problémem jsou proměnné. Těm se (v PL1) prostřednictvím valuace přiřazují individua, tj. prvky universa. Jak dále uvidíme, hodnota formule nezávisí na hodnotě vázaných proměnných (pouze volné proměnné jsou „skutečné“ proměnné). Obsahuje-li formule nějaké volné proměnné, můžeme vyhodnotit její pravdivost při interpretaci pouze v závislosti na ohodnocení (valuaci) volných proměnných. Existují interpretace, ve kterých mohou být formule v daných interpretací pravdivé, při jiných nepravdivé.

Pro úplnost pochopení dané problematiky je níže uvedena definice interpretace formulí (jazyka PL1).

2.1.5 Definice - Interpretace

Interpretace jazyka predikátové logiky 1. řádu je tato trojice objektů (někdy nazývána interpretační struktura):

- Neprázdná množina M, která se nazývá universum diskursu a její prvky jsou individua.
- Interpretace funkčních symbolů jazyka, která přiřazuje každému n-árnímu funkčnímu symbolu f určité zobrazení (totální) f_M: $M^n \rightarrow$M.
- Interpretace predikátových symbolů jazyka, která přiřazuje každému n-árnímu predikátovému symbolu p jistou n-ární relaci p_M nad M, tj. podmnožinou Kartézského součinu M^n.

Tolik k samotné definici. Uvedeme si ještě další poznámky, které napomohou k lepšímu pochopení této definice.

2.1.6 Poznámky - Interpretace

1. Každý n-ární funkční symbol je tedy interpretován jako funkce, která přiřazuje n-tici individuí přávě jedno individuum, tj. zobrazení z M x...x M do M. Speciálně:
    - je-li n = 0, pak se jedná o nulární funkční symbol, tedy o individuovou konstantu, které je přiřazen prvek universa - individuum

- je-li $n = 1$, pak se jedná o unární funkční symbol, kterému je přiřazena funkce o jednom argumentu (např. nad množinou čísel $x^2, x + 1$, nad množinou individuí otec(x), matka(x), atd)
- je-li $n = 2$, pak se jedná o binární funkční symbol, kterému je přiřazena binární funkce se dvěma argumenty.

2. Každý n-ární predikátový symbol p je je interpretován jako n-ární relace p_M tj. podmnožina Kartézského součinu M x...x M, neboli zobrazení M x...x M $\rightarrow$ {1,0}. Tato relace p_M se nazývá obor pravdivosti predikátu. Specielně:

   - je-li $n = 0$, pak se jedná o nulární predikátový symbol, kterému je přařazena hodnota 1 nebo 0 (pravda, nepravda) tak, jak to jež známe z výrokové logiky.
   - je-li $n = 1$, pak se jedná o unární predikátový symbol, kterému je přařazena podmnožina universa M.
   - je-li $n = 2$, pak se jedná o binární predikátový symbol, kterému je přiřazena binární relace nad universem.

V tuto chvíli již víme jak interpretaci chápat. Jak provádět valuaci (ohodnocení individuových proměnných) si definujeme v další sekci.

2.1.7 Definice - Valuace

Ohodnocení (valuace) individuových proměnných je zobrazení e, které každé proměnné x přiřazuje hodnotu $e(x) \in M$ (prvek univerza)

Ohodnocení termů e^* indukované ohodnocením proměnných e je induktivně definováno takto:

- $e^*(x) = e(x)$
- $e^*(f(t_1, t_2, ..., t_n)) = f_M(e^*(t_1), e^*(t_2), ..., e^*(t_n))$, kde f_M je funkce přiřazená v dané interpretaci funkčnímu symbolu f.

Již víme jak formule chápat a jak je ohodnocovat - zadefinujeme si tedy, co je pravdivost formule a jak o ní uvažovat.

2.1.8 Základní definice

Formule A je splnitelná v interpretaci I, jestliže existuje ohodnocení e proměnných takové, že platí $| =_I A[e]$.

Formule A je pravdivá v interpretaci I, značíme $| =_I A$, jestliže pro všechna možná ohodnocení e individuových proměnných platí, že $| =_I A[e]$.

Model fomule A je interpretace I, ve které je A pravdivá.

Formle A je splnitelná, jestliže existuje interpretace I, ve které je pravdivá, tj. jestliže existuje interpretace I a valuace e takové, že $| =_I A[e]$.

Formule A je tautologií (logicky pravdivá), značíme $| = A$, jestliže je pravdivá v každé interpretaci.

Formule A je kontradikcí, jestliže nemá model, tedy neexistuje interpretace I, která by formuli A splňovala.

Model množiny formulí $\{A_1, ..., A_n\}$ je taková intrepretace I, ve které jsou pravdivé všechny formule $A_1, ..., A_n$.

Formule B logicky vyplývá z formulí $A_1, ..., A_n$, značíme $A_1, ..., A_n| = B$, jesltiže B je pravdivá v každém modelu množiny formulí $A_1, ..., A_n$. Tedy pro každou interpretaci I, ve které jsou pravdivé formule $A_1, ..., A_n$ ($| =_I A_1, ..., A_n$) platí, že je v ní pravdivá také formule B($| =_I$B).

Nyní je pravá chvíle na to, abychom si ukázali některé sémantické důkazové metody. Slovem některé máme na mysli metodu Vennových diagramů a metodu relačních struktur. Obě jsou probrány v následující kapitole i s ukázkovými příklady.

2.2 Sémantické důkazové metody - Vennovy diagramy a relační struktury

Sémantické metody se zajímají sémantikou, tedy významem. V PL1 je významem interpretace. Metoda **Vennových diagramů** je grafická důkazová metoda, která je použitelná pro formule s jednoargumentovými predikáty. Algoritmus této metody je následující: Obory pravdivosti predikátů zakreslíme jako vzájemně se protínající se kroužky (každé dvě množiny - kroužky - mají společný průnik), znázorníme situaci, kdy jsou premisy pravdivé - tj: vyšrafujeme plochy, které odpovídají prázdným třídám objektů (všeobecné předpoklady) a označíme křížkem plochy, které jsou neprázdné (existenční předpoklady); křížek přitom klademe jen tehdy, když neexistuje jiná plocha, „kam by mohl přijít". Nakonec ověříme, zda vzniklá situace znázorňuje pravdivost závěru.

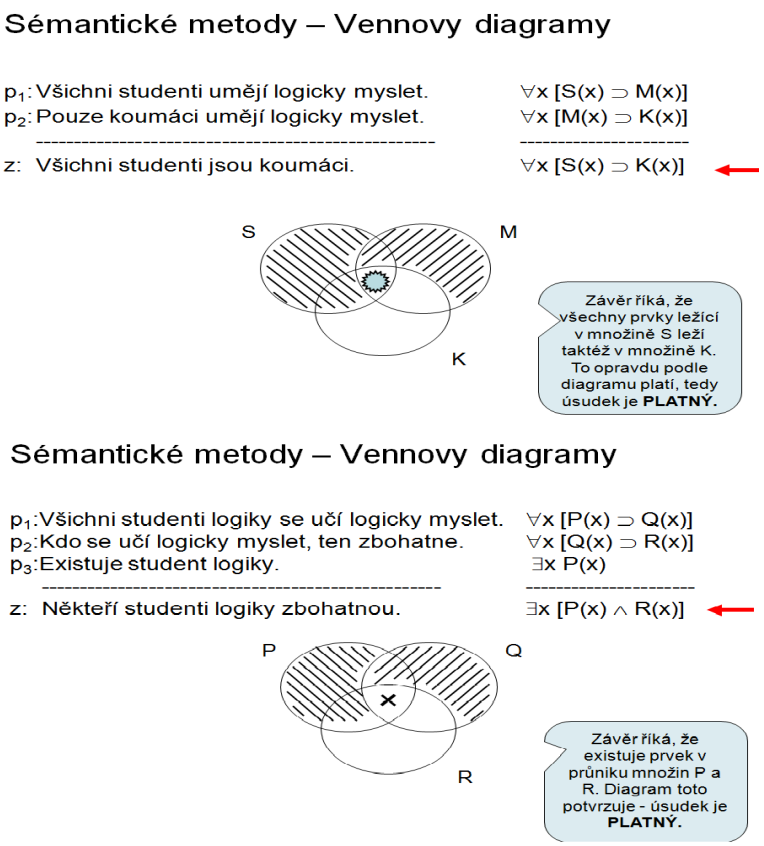
Příkldy věnující se metodám Vennových diagramů můžeme vidět na obrázku 1.

Metoda **relačních struktur** může narozdíl od Vennových diagramů pracovat i s víceargumentovými predikáty. Metoda relačních struktur už nevyužívá kroužků ke znázornění množin, ale využívá relací a úvah o vztazích mezi prvky těchto relací. Více o této metodě nám určitě ukáže příklad řešený na obrázku 2.

Těmito řádky ukončíme sémantické metody a dalšími metodami, kterými se budeme zabývat jsou metody syntaktické. V dalších kapitolách se budeme věnovat hlavně Rezoluční metodě - syntaktická metoda. Uvedemi si nejprve teoretické základy a pak nějaké příklady, na kterých teorii použijeme.

2.3 Automatické dokazování v predikátové logice (obecná rezoluční metoda)

Rezoluční metoda je jedna z procedur (algoritmů), které parciálně rozhodují, zda daná formule PL1 je nesplnitelná. Pro předloženou formuli A, která nesplnitelná je, tedy procedura v konečném čase tuto skutečnost zjistí a zastaví se. V případě, že A je splnitelná, algoritmus nemusí nikdy skončit svou činnost. Chceme-li tedy rozhodnout, zda daná formlue A je logicky pravdivá, použijeme rezoluční metodu na formuli ¬A a zjišťujeme, zda je nesplnitelná. Je-li tomu tak, procedura to zjistí a vydá kladnou odpověď. V opačeném případě proces nemusí nikdy skončit. Specielně, chceme-li zjistit, zda $\{A_1, ..., A_n\} | = B$,



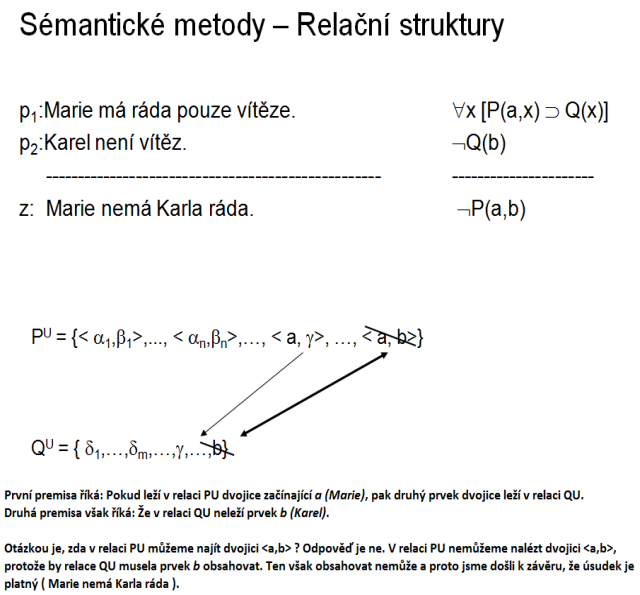
Obrázek 1: Vennovy diagramy - příklady.

aplikujeme rezoluční metodu na formuli $A_1 \wedge ... \wedge A_n \wedge \neg B$, neboť pokud je tato formule nesplnitelná, pak je formule $(A_1, ..., A_n) \supset$ B tautologie a vztah vyplývání platí.

Rezoluční metodu lze aplikovat pouze na formule v klauzulární (Skolemově) formě. Každá formule je převoditelná do klauzulární formy tak, že výsledná formule je splnitelná, právě když výchozí fomule je splnitelná.

Automatické dokazování v predikátové logice zobecňuje postupy automatického dokazování výrokové logiky. Oproti situaci ve výrokové logice je situace v predikátové logice složitější z těchto důvodů:

- Komplikovanější je procedura převedení formule na klauzulární tvar. Oproti výrokové logice obsahuje navíc:
  - eliminaci kvantifikátorů z formule,
  - převod formule na prenexní tvar.
- Složitější je tvar rezolučního odvozovacího pravidla. Jeho použití vyžaduje simultánní substituci.



Obrázek 2: Relační struktury - příklady.

Jak již bylo řečeno, pro aplikování rezoluční metody musíme mít formuli ve skolemově formě - proto se nyní podíváme na to, co to skolemova forma je a uvedeme její definici.

2.3.1 Definice - Skolemova forma

Skolemova forma uzavřené formule je prenexní tvar této formule, která neobsahuje žádné existenční kvantifikátory. Skolemova forma vznikne z prenexní formy opakovaným použitím následujících dvou operací (skolemizací):

$\forall x_1\ \forall x_2..\forall x_n \exists y\ \mathrm{A}(x_1,x_2,..,x_n,\mathrm{y}) \rightarrow \forall x_1\ \forall x_2..\forall x_n \mathrm{A}(x_1,x_2,..,x_n,\mathrm{f}(x_1,x_2,..,x_n))$

kde f je nový (v jazyce dosud nepoužitý) n-ární funkční symbol, tzv. Skolemova funkce,

$\exists \mathrm{x} \forall y_1 \forall y_2,..,\forall y_n \mathrm{A}(\mathrm{x},y_1,y_2,..,y_n) \rightarrow \forall y_1 \forall y_2,..,\forall y_n \mathrm{A}(\mathrm{c},y_1,y_2,..,y_n)$,

kde c je nová (v jazyce dosud nepoužitá) individuová konstanta, tzv. Skolemova konstanta;

Každému eliminovanému existenčnímu kvantifikátoru odpovídá jiná Skolemova konstanta.

Skolemovu formu formule A označíme zápisem A^S.

Literál je atomická formule nebo negace atomické formule (např. p(f(x)),¬q(y)).

Klausule je disjunkce literálů (např. [p(f(x)) ∨ ¬q(y)]).

Konjunktivní normální tvar formule predikátové logiky je prenexní tvar formule, jejíž matice je konjunkce disjunkcí literálů (tj. konjunkce klauzulí).

Disjunktní normální tvar formule predikátové logiky je prenexní tvar formule, jejíž matice je disjunkce konjunkcí literálů.

Skolemova forma formule je klausulární forma, jejíž matice je v klauzulárním tvaru, tj. je konjunkcí klausulí.

2.3.2 Věta - Skolem

Každá formule A může být převedena na formuli A^S v klauzulární (Skolemově) formě takovou, že A^S je splnitelná, právě když A je splnitelná.

2.3.3 Algoritmus - Skolem

Uvedli jsme si co je rezoluční metoda, s jakými formulemi pracuje a nyní se podíváme na algoritmus, kterým lze každou formuli převést do její skolemovy formy.

1. Utvoření existenčního uzávěru formule A (Zachovává splnitelnost)
2. Eliminace nadbytečných kvantifikátorů. (Ekvivalentní krok)

   Z formule A vypustíme všechny kvantifikátory $\forall x_i, \exists x_i$, v jejichž rozsahu se nevyskytuje proměnná x_i.
3. Přejmenování proměnných (Ekvivalentní krok)

   Přejmenujeme všechny proměnné, které jsou v A kvantifikokvány více než jednou tak, aby všechny kvantifikátory měly navzájem různé proměnné.
4. Eliminace spojek $\supset$, $\equiv$ podle těchto vztahů (Ekvivalentní krok)

   (A $\supset$ B) $\Leftrightarrow$ ($\neg$A $\vee$ B), (A $\equiv$ B) $\Leftrightarrow$ ($\neg$A $\vee$ B) $\wedge$ ($\neg$B $\vee$ A)
5. Přesun spojek $\neg$ dovnitř. (Ekvivalentní krok)
6. Přesun kvantifikátorů doprava. (Ekvivalentní krok)

   Provádíme náhrady podle těchto ekvivalencí (Q je kvantifikátor $\forall$ nebo $\exists$; Θ je symobl $\wedge$ nebo $\vee$; A, B neobsahují volnou proměnnou x):

   Qx(A Θ B(x)) $\Leftrightarrow$ A Θ Qx B(x), Qx(A(x) Θ B) $\Leftrightarrow$ Qx A(x) Θ B

   Pozn. Před provedením kroku 7 je vhodné provést ekvivalentní zjednodušující úpravy formule.
7. Eliminace existenčních kvantifikátorů (Zachovává splnitelnost)

   Provádíme postupně Skolemizaci podformulí Qx B(x), Qx A(x), které jsme obdrželi v předchozím kroku 6 (viz. kapitola 2.3.1).
8. Přesun všeobecných kvantifikátorů doleva (Ekvivalentní krok, neboť jsme již provedli krok 3. a platí ekvivalence dle 6)

9. Použití distributivních zákonu (Ekvivalentní krok)

   Provedeme postupné náhrady vlevo formulemi vpravo:

   (A ∧ B) ∨ C ⇔ (A ∨ C) ∧ (B ∨ C), A ∨ (B ∧ C) ⇔ (A ∨ B) ∧ (A ∧ C)

Pro řešení otázek ohledně pravdivosti formulí je skolemizace jen jedním podkrokem. Celý proces který rozhoduje o pravdivosti se nazývá Herbrandova procedura.

2.3.4 Herbrandova procedura

Chceme-li dokázat ligickou pravdivost formule A v PL1, pak budeme postupovat obdobně jako ve VL:

1. Formuli A znegujeme.
2. Formuli ¬A převedeme do klausulární formy $(\neg A)^S$
3. Na formuli $(\neg A)^S$ budeme postupně uplatňovat resoluční pravidlo. Pokud získáme prázdnou klausuli #, důkaz je ukončen

Bod 3 není až tak lehce proveditelný, neboť literály s opačným znaménkem, které bychom mohli při uplatňování rezoluce „vyškrtávat“, mohou obsahovat různé termy.

Musíme, tedy použít vhodné substituce, abychom mohli korektně literály vyškrtávat. Pro nalezení vhodných substitucí můžeme použít určité metody - uvedu dvě:

- procedura hledání Herbrandova univerza,
- Robinsonův unifikační algoritmus.

Herbrandova procedura parciálně rozhoduje, zda je daná formule A nesplnitelná. K dané formuli postupně generujeme základní instance jejích klausulí a resoluční metodou vždy testujeme, zda je jejich konjunkce nesplnitelná. Jestliže tomu tak je, pak A je nesplnitelná a tato procedura to po konečném počtu kroků zjistí. V případě splnitelnosti A může procedura generovat donekonečna nové a nové instance a testovat jejich konjunkce.

Efektivita tohoto algoritmu není příliš velká a v samotném řešení se tento algoritmus nepoužívá. Mnohem jednodušším algoritmem je Robinsonův algoritmus. Robinsonův algoritmus pro hledání substitucí, které unifikují některé literály a umožní dokázat nesplnitelnost, bude popsán v dalších kapitolách.

Pro teoretický základ k Robinsonovu algoritmu si uvdeme definice pro instanci a unifikaci.

2.3.5 Definice - Instance

Nechť A je formule obsahující individuové proměnné x_i, i=1,2,...n a to buď přímo (jako bezprostřední argumenty) nebo zprostředkovaně (jako argumenty funkcí). Označme

$\delta = \{x_1/t_1, x_2/t_2, ..., x_n/t_n\}$

simultánní substituci termů t_i za (všechny výskyty) přdmětové proměnné x_i pro i = 1,2,...,n. Potom zápisem

Aδ

označíme formuli, která vznikne z formule A provedením substutuce δ. Poznamenejme, že substituce se může týkat všech, nebo jen některých, nebo dokonce žádné individuové proměnné obsažené v A (v tomto případě pro některá nebo všechna i substituujeme x_i/x_i).

Formule B je instancí formule A, jestliže existuje substituce δ taková, že B = Aδ.

Poznámka: Substituce lze skládat. Pro skládání substitucí platí:

- $(\delta\rho)\tau = \delta(\rho\tau) = \delta\rho\tau$, tj. skládání substitucí je asociativní.
- Pro identickou substituci (tj. x_i/x_i pro všechna i)ε platí $\varepsilon\delta = \delta\varepsilon = \delta$, tj. identická substituce hraje v algebře substitucí úlohu jednotkového prvku.
- $\delta\rho \neq \rho\delta$, tj. skládání substitucí není obecně komutativní.

2.3.6 Definice - Unifikace

Unifikace (unifikační substituce, unifikátor) formulí A, B je substituce δ taková, že

Aδ = Bδ.

Nejobednější unifikace formulí A, B je unifikace δ taková, že pro každou jinou unifikaci ρ formulí A, B platí $\rho = \delta\tau$, kde $\tau \neq \varepsilon$, tj. každá unifikace vznikne z nejobecnější unifikace provedením další dodatečné substituce.

2.3.7 Algoritmus nalezení nejobecnější unifikace - Robinson

V roce 1965 J. A. Robinson navrhl nový unifikační algoritmus, který narozdíl od Herbrandovy procedury nevyžaduje generování základních instancí, ale rozhoduje přímo, zda k libovolné konjunkci klausulí existuje substituce taková, která unifikuje některé literály a umožní dokázat nesplnitelnost (pokud tato konjunkce nesplnitelná je).

Formulace zcela obecného algoritmu je poměrně složitá (patří do výpočeních metod umělé inteligence) a jeho „ruční“ simulace značně nepřehledná. Omezíme se proto pouze na případ, kdy unifikované elementární formule nemají na obou místech stejnolehlých argumentů současně nějaké funkční struktury.

Předpokládejme tedy

A = p(t_1,t_2,...,t_n), B = p(s_1,s_2,...,s_n)

kde t_1,t_2,...,t_n, s_1,s_2,...,s_n jsou termz takové, že t_i,s_i nejsou současně funkční struktury (tedy alespoň jeden z nich je proměnná). Potom nejobecnější unifikaci získáme takto:

1. Pro i = 1,2,...,n prováděj:
   - Je-li $t_i = s_i$, pak polož $\delta_i = \varepsilon$.
   - Není-li $t_i = s_i$, pak zjisti, zda jeden z termů t_i, s_i představuje nějakou individuovou proměnnou x a druhý nějaký term r, který proměnnou x neobsahuje.

- Jestliže ano, pak polož $\delta_i = \{x/r\}$.
- Jestliže ne, pak ukonči práci s tím, že formule A, B nejsou unifikovatelné.

2. Po řádném dokončení cyklu urči $\delta = \delta_1\delta_2...\delta_n$. Substituce δ je nejobecnější unifikací formulí A, B.

2.3.8 Obecné rezoluční pravidlo

Nechť A_i, B_i, L_i jsou atomické formule predikátové logiky. Potom platí následující odvozovací pravidlo:

$A_i \vee ... \vee A_m \vee L_1$, $B_1 \vee ... \vee B_n \vee \neg L_2 \mid\!- A_i\rho \vee ... \vee A_m\rho \vee B_1\rho \vee ... \vee B_n\rho$, kde ρ je unifikace formulí L_1, L_2, tj. $L_1\rho = L_2\rho$.

Klauzule na levé straně odvozovacího pravidla nazýváme rodičovskými klauzulemi a klauzuli na prvé straně rezolventou.

Formule A^S v klausulární formě je nesplnitelná, právě když z ní lze opakovaným použitím obecného pravidla rezoluce odvodit prázdnou klausuli o.

Výše popsané pravidlo je uplatňováno v poslední části procedury rozhodující o pravdivosti zadané formule. Pro úplné pochopení rezoluční metody ještě přikládám názorný příklad na obrázku 3.

2.4 Systém přirozené dedukce predikátové logiky

2.4.1 Úvodní poznámky

Metoda přirozené dedukce predikátové logiky je zobecněním metody přirozené dedukce výrokové logiky. Systém využívá rozšířenou množinu výchozích dedukčních pravidel. Tato pravidla jsou vypsána v další kapitole.

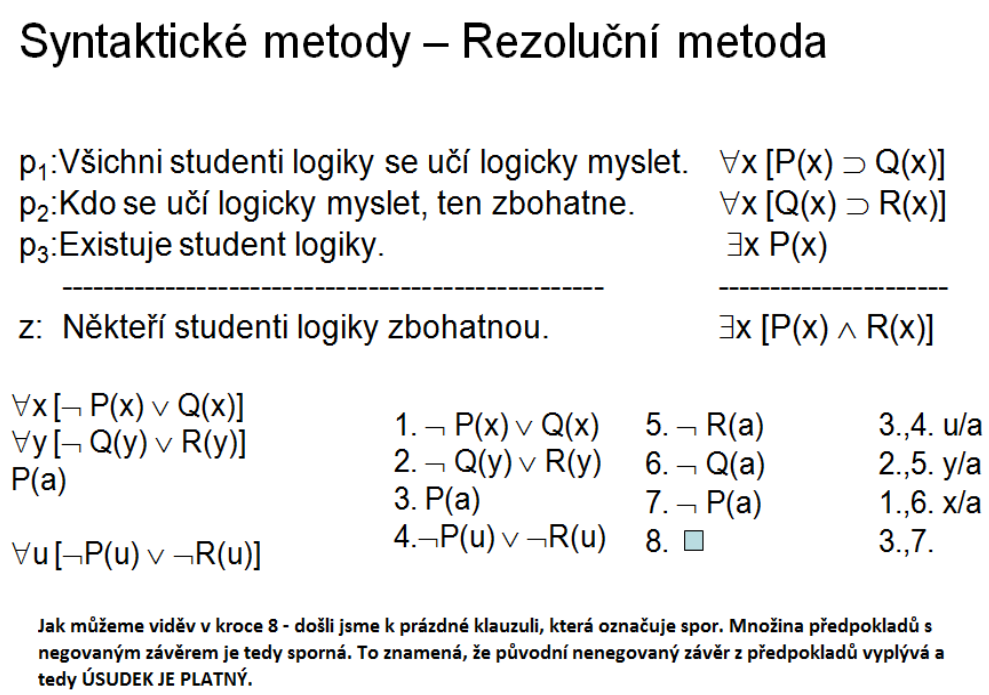
2.4.2 Definice - Dedukční pravidla

Výchozími (nedokazovanými, primárními) dedukčními pravidly jsou všechna následující pravidla:

Zavedení konjunkce:	A,B $\mid\!-$ A $\wedge$ B	ZK
Eliminace konjunkce:	A $\wedge$ B $\mid\!-$ A,B	EK
Zavedení disjunkce:	A $\mid\!-$ A $\vee$ B nebo B $\mid\!-$ B $\vee$ B	ZD
Eliminace disjunkce:	A $\vee$ B,$\neg$A $\mid\!-$ B nebo B $\vee$ A,$\neg$B $\mid\!-$ A	ED
Zavedení implikace:	B $\mid\!-$ A $\supset$ B	ZI
Eliminace implikace:	A $\supset$ B,A $\mid\!-$ B	EI - MP
Zavedení ekvivalence:	A $\supset$ B, B $\supset$ A $\mid\!-$ A $\equiv$ B	ZE
Eliminace ekvivalence:	A $\equiv$ B $\mid\!-$ A $\supset$ B,B $\supset$ A	EE
Zavedení obecného kvantifikátoru:	A(x) $\mid\!-$ $\forall$xA(x)	Z$\forall$

Pravidlo lze použít pouze tehdy, jestliže formule A(x) není odvozena z žádného předpokladu, který obsahuje x jako volnou proměnnou.

Eliminace obecného kvantifikátoru: $\forall$xA(x) $\mid\!-$ A(x/t) E$\forall$



Obrázek 3: Rezoluční metoda - příklad.

Forumle A(x/t) je výsledkem korektní substituce termu t za proměnnou x ve formuli A(x), tedy term T musí být substituovatelný za x ve formuli A.

Zavedebá existenčního kvantifikátoru: $A(x/t) \vdash \exists x A(x)$ Z∃

Eliminace existenčního kvantifikátoru: $\exists x A(x) \vdash A(x/c)$ E∃

Použijeme-li pravidlo E∃ pro různé formule A, musíme za proměnnou x substituovat vždy jinou konstantu c.

Obsahuje-li formule A, kromě vantifikované proměnné x, ještě další volné proměnné, lze pravidlo eliminace existenčního kvantifikátoru formulovat obecněji takto:

$\exists x A(x, y_1, ... y_n) \vdash A(x \,/\, f(y_1, ..., y_n), y_1, ..., y_n)$

V tomto případě nelze za kvatifikovanou proměnnou x substituovat konstantu, ale funkci zbývajících (volných) proměnných. Použijeme-li pravidlo vícekrát pro různé formule A, musíme za proměnnou x substituovat vždy jinou funkci $f(y_1, ..., y_n)$.

V dalších dvou krátkých kapitolkách jsou fakty, které je důležité si uvědomit.

2.4.3 Věta - O korektnosti

Každý toerém (dokazatelná formule) systému přirozené dedukce predikátové logiky je tautologií predikátové logiky: Jestliže $|- A$, pak $| = A$.

2.4.4 Věta - O sémantické úplnosti

Každá tautologie predikátové logiky je v systému přirozené dedukce dokazatelná (je teorémem): Jestliže $| = A$, pak $|- A$.

2.5 Axiomaticý systém predikátové logiky

2.5.1 Úvodní poznámky

Axiomatická metoda v predikátové logice je zobecněním axiomatické metody ve výrokové logice. Od té se liší tím, že pracuje s obecnějším jazykem a v souvislosti s tím používá rozšířenou množinu výchozích teorémů (axiómů, resp. axiomových schémat) a rozšířenou množinou výchozch odvozovacích pravidel - viz náseldující definice. Přímočaře se zobecňují hlavní věty o axiomatickém systémuvýrokové logiky: věta o dedukci, věta o korektnosti a sémantické úplnosti.

2.5.2 Formální systém (logický kalkul) Hilbertova typu

Definice - Axiomatický systém Hilbertova typu.

- Jazyk: viz definice 2.4.2 s jednou výjimkou: množina funktorů je omezena na funktory $\neg, \supset, \forall$.
- Axiómová schémata:
  - $A \supset (B \supset A)$
  - $(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$
  - $(\neg B \supset \neg A) \supset (A \supset B)$
  - $\forall x A(x) \supset A(x/t)$ Term je substituovatelný za x v A
  - $(\forall x[A \supset B(x)]) \supset (A \supset \forall x B(x))$, x není volná proměnná v A
- Odvozovací pravidla
  - $A, A \supset B |- B$ (modus ponens)
  - $A |- \forall x A$ (pravidlo generalizace)

Poznámky k definici:

1. Definovaný axiomatický systém pracuje pouze s funktory $\neg$, $\supset$, $\forall$. Ostatní funktory můžeme používat jako zkratky definované takto:

Z1: $A \wedge B = {}_d f \neg(A \supset B)$
Z2: $A \vee B = {}_d f \neg A \supset B$
Z3: $A \equiv B = {}_d f (A \supset B) A \wedge B (B \supset A)$
Z4: $\exists x A = {}_d f \neg \forall x \neg A$

Symboly $\wedge$, $\vee$, $\equiv$, $\exists$ nepatří do jazka definovaného axiomatického systému, jsou to metasymboly sloužící k označování složených formulí jistého typu.

Axiomatických systémů predikátové logiky je mnoho a různé systémy pracují s různými množinami funktorů.

2. Volba axiómů není pochopitelně zcela libovolná; aby byl systém korektní, podlehá dvěma kritériím:

   - Každý axióm je tautologie.
   - Množina axiómů musí umožňovat, aby se z nich daly odvodit všechny logikcy platné formule a přitom by měla být množina axiómů nezávislá.

3. Rovněž volba odvozovacích pravidel není libovolná. Aby byl systém korektní, musí pravidla zachovávát pravdivost v tom smyslu, že formule, kterou podle pravidla obdržíme, je pravdivá alespoň ve všech modelech předpokladů pravidla, tedy z předpokladů vyplývá.

Jak bylo uvedeno výše, axiomatických systémů je plno. Jako příklad si můžeme v rychlosti představit **Gentzenův systém přirozené dedukce (Gentzenovský výrokový kalkul)**, který vychází pouze z jednoho axiómu - $A \supset A$ neboli $A \vee A$. Dedukční pravidla jsou podobná jako v polském systému přirozené dedukce.

Následují tři krátké věty, které jsou však velice důležité svým obsahem a je nutno je pochopit a uvědomit si, co vlastně znamenají.

2.5.3 Věta - O dedukci

Pro uzavřenou formuli A a libovolnou formuli B platí:

- $\vdash$ A $\supset$B právě tehdy, když A $\vdash$ B (syntaktická podoba)
- $\models$ A $\supset$B právě tehdy, když A $\models$ B (sémantická podoba)

2.5.4 Věta - O korektnosti

Každá dokazatelná formule predikátové logiky (tj. teorém kalkulu Hilbertova typu) je také tautologií predikátové logiky. Formálně: jestliže $\vdash$ A, pak $\models$ A.

2.5.5 Věta - O sémantické úplnosti axiomatického systému - K. Gödel

Každá tautologie predikátové logiky je dokazatelná (v logickém kalkulu Hilbertova typu). Formálně, je-li $\models$ A pak $\vdash$ A.

2.6 Shrnutí

Po přečtení kapitoly 2 bychom měly mít přehled o:

- Jazyku predikátové logiky
- Volné a vázané proměnné, uzavřenosti formule
- Interpretaci jazyka
- Valuaci
- Pravdivosti formule
- Splnitelnosti fromule
- Sémantických důkazových metodách (Vennovy diagramy, relační struktury)
- Obecné rezoluční metodě
- Herbrandově proceduře
- Instanci
- Unifikaci
- Robinsnově algoritmu
- Obecném rezolučním pravidlu
- Systému přirozené dedukce predikátové logiky
- Axiomatickém systému predikátové logiky

3 Popis řešení

Poznámka: V následujících odstavcích je popsáno, jak se autor postupně vyrovnával s problémy, které řešení obsahuje. Aby shrnující popis nebyl příliš komplikovaný, je použito zobecnění. Přehledný a detailní popis všech klíčových funkcí programu je uveden v dalších podkapitolách.

Hlavním cílem této diplomové práce bylo vytvoření modulu do již existujícího systému, který uživatele krok po kroku provádí procesem dokazování platnosti formule pomocí obecné rezoluční metody, napovídá a také jednotlivé kroky vyhodnocuje.

Jako **vstup** pro vytvořený modul byl určen soubor, který obsahuje seznam zadání (formulí) v podobě řetězců. Byla také stanovena **jednoznačná pravidla**, která musí každá formule zadání dodržovat (aby program s touto formulí správně pracoval).

Většina kroků, které obecná rezoluční metoda obsahuje, potřebuje pracovat se **stromovými reprezentacemi formule**.

Bylo tedy nejprve nutno nalézt způsob, jak formuli reprezentovanou řetězcem převést na formuli reprezentovanou **binárním stromem**. Mezi další nutné vnitřní funkce modulu jednoznačně zapadla funkce **kontroly zadaného vstupu pomocí specifikované gramatiky**, aby nedocházelo k neočekávaným selháním celého modulu.

Po vyřešení syntaktické analýzy formulí přišly na řadu otázky spojené s **převodem formule do klausulární formy**, do které musí být formule převedena, pakliže chceme dokazovat, že je daná formule tautologií (respektive, že závěr vyplývá z předpokladů). Proces převodu vyústil v sadu desíti úprav (pravidel), které jsou na formuli postupně (dle dohodnutých scénářů) uplatňovány a je tak docíleno samotného převodu formule do klausulární formy.

Po úspěšném vyřešení problému převodu formule do klausulární formy nastal problém s **extrakcí klausulí** a s tím spojený proces **rezoluce**, který rekurzivně (**s dohodnutou hranicí maximálního počtu rekurzicních zanoření**) zkouší nalézt takovou sadu uplatnění pravidla rezoluce vždy na právě dvě klausule, která povede k **nalezení sporné situace** (sporu). Problémy extrakce klausulí i rezoluce byly úspěšně vytřešeny.

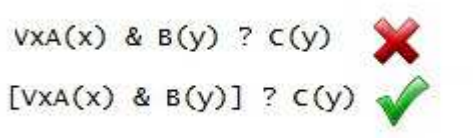
Posledním stěžejním blokem byl **systém nápovědy**. Nápověda uživateli radí v úpravách formule do klausulární formy, v extrakci klausulí a také v rezoluci. Nápověda byla také vyřešena.

3.1 Pravidla pro zadávání formulí, speciální znaky versus vnitřní znaky programu

Program pro definice formulí používá následující **pravidla** (formule zadává například uživatel při práci s programem):

1. Predikáty musí začínat písmeny: A B C D F G H I J K L M.
2. Konstanty musí začínat písmeny: N O P Q R S T U W X Y Z.
3. Funkce musí začínat písmeny : a b c d e f g h i j k l m.
4. Proměnné musí začínat písmeny : n o p q r s t u v w x y z.

5. Za prvním symbolem predikátu, konstanty, funkce a proměnné se může nalézat libovolně dlouhý číselný index (složený jen z číslic).
6. Ve formuli se můžou používat mezery (ale jsou před operacemi s formulí odstraněny a výsledky operací nad formulí jsou prozentovány uživateli bez mezer).
7. Všeobecný kvantifikátor se značí písmenem: V.
8. Existenční kvantifikátor se značí písmenem: E.
9. Ve formuli se mohou používat závorky: (), [], {} (všechny závorky jsou před operacemi s formulemi nahrazeny za závorky () a výsledky operací nad formulí jsou uživateli prezentovány jen se závorkami typu ()).
10. Pro oddělení argumentů v závorkách se používá symbol „ , “.
11. Logickou spojku konjunkce zapisujeme symbolem „&“.
12. Logickou spojku disjunkce zapisujeme symbolem „ | “.
13. Logickou spojku implikace zapisujeme symbolem „?“.
14. Logickou spojku ekvivalence zapisujeme symbolem „ ~“.
15. Posledním a nejdůležitějším pravidlem je správné uzávorkování formule. Každé dvě binární spojky musí být odděleny závorkou - viz. obrázek 4.



Obrázek 4: Špatné uzávorkování

Znaky jako jsou V, E, ? apod. jsou takzvané **vnitřní znaky programu**. S těmito znaky se při operacích, jako jsou například operace vedoucí k převodu formule do klausulární formy, přímo pracuje. Důvod je jednoduchý, dají se pohodlně ukládat do vnitřních proměnných programu, dájí se pohodlně porovnávat a jejich zápis je jednoduchý.

Jediným problémem vnitřních znaků programu je, že se v logice používají pro smyslem stejné výrazy jiné symboly. Například všeobecný kvantifikátor (v porgramu V) je v logice značen symbolem ∀. Proto pokud jsou uživateli prezentovány jakékoliv formule, musí se v nich vnitřní znaky nahradit znaky, které jsem v této práci označil jako **znaky speciální**.

Následující tabulka znázorňuje všechny důležité vnitřní znaky programu s jejich jiným ekvivalentním zápisem (speciálním zápisem).

Konjunkce	&	$\wedge$
Disjunkce	|	$\vee$
Implikace	?	$\supset$
Ekvivalence	~	$\equiv$
Negace	!	$\neg$
Všeobecný kvantifikátor	V	$\forall$
Existenční kvantifikátor	E	$\exists$

3.2 Hlavní případ užití programu

Prvním krok, který musí uživatel provést, je **vygenerování zadání** (pomocí tlačítka generuj - viz.kapitola 5). Program načte a zobrazí náhodně vybranou formuli ze souboru všech formulí, současně zpřístupní uživateli funkce spojené s úpravou formule do klausulární formy. Uživatel za doprovodu kontroly programu **upravuje formuli až do klausulární formy**. Posledním krokem před **rezolucí** je **extrakce klausulí** do jednotlivých řádků (jeden řádek odpovídá jedné klausuli).

Pokud uživatel dojde v procesu rezoluce ke sporu, pak program ukončí svoji činnost a znemožní všechny funkce krom generování nového zadání (s generováním nového zadání se program opět nastaví do stavu aktivního napovídání a je schopen vést uživatele celým procesem od začátku).

3.3 Schéma fungování programu

V této podkapitole je popsáno, jak program funguje v rámci provádění jednoho kroku (krokem je zde myšleno například: provedení jedné úpravy formule vedoucí ke klausulární podobě formule) před extrakcí klausulí, během extrakcí klausulí a také během uplatňování rezolučního pravidla.

3.3.1 Schéma provádění kroku - úpravy do klausulární formy

- Nejprve se nahradí speciální znaky formule za znaky, se kterými se dá lépe pracovat (dále jen vnitřní znaky).
- Provede se syntaktická analýza formule.
- Provede se převod formule ze stringové reprezentace na stromovou reprezentaci.
- Najde se a provede vhodná úprava (např.: přejmenování proměnných, přesun negace doprava apod.) formule (strojem).
- Provedená úprava se porovná s úpravou, která se zdá uživateli jako aktuálně nutná k provedení (to znamená, že se porovná, zda uživatel vybral úpravu, kterou stroj provedl).
- Pokud uživatel zvolil úpravu, která se nad formulí provedla, pak dochází ke srovnání formule, na kterou program aplikoval příslušné pravidlo s formulí, kterou

zadal uživatel (formule se srovnávají ve stromových reprezentacích, jak je uvedeno dále v kaptitole 3.14.2). Po srovnání dochází ke zhodnocení celého kroku.

- Zpětný převod formule ze stromové reprezentace na stringovou reprezentaci a vizualizace výsledků.

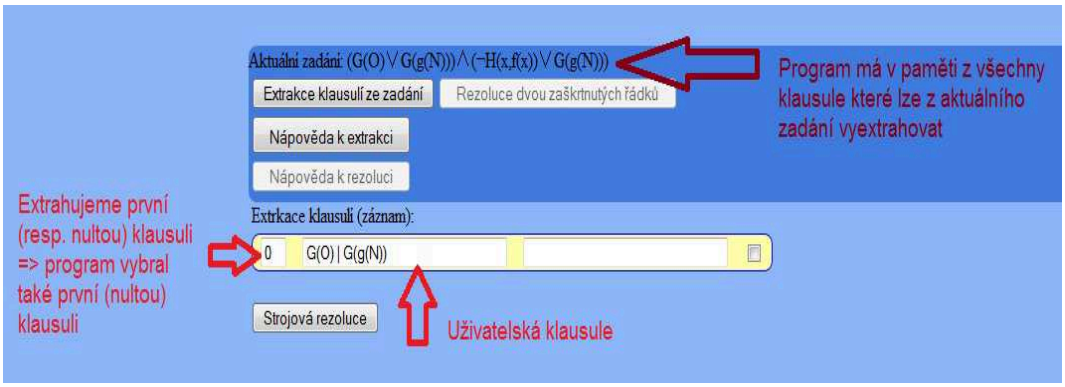
Poznámka: Tato podkapitola jen naznačuje, jak program uplatňuje na formuli úpravy týkající se převodu formule do klausulární formy. Detaily jednotlivých kroků schématu (algoritmy apod.) jsou dále v textu vysvětleny.

3.3.2 Schéma provádění kroku - extrakce klausulí

Nutno podotknout, že při provádění jednoho kroku (extrakce jedné klausule ze zadání) má program v paměti všechny klausule, které je možno (nutno) ze zadání extrahovat (ve standardním pořadí jejich extrakce zleva doprava). Každý krok pak vypadá následovně:

- Program vybere klausuli (má v paměti všechny extrahované klausule), která je v procesu extrakce na řadě a provede u této klausule náhradu speciálních znaků na vnitřní znaky.
- V uživatelské klausuli se také nahradí všechny vnitřní znaky za znaky speciální.
- Program provede porovnání a vyhodnocení obou klausulí.

Poznámka: Tato podkapitola jen naznačuje, jak program provádí extrakci jedné klausule ze zadání. Detaily jednotlivých kroků schématu (algoritmy apod.) jsou dále v textu vysvětleny. Pro přesnější představu o schématu provedení extrakce jedné klausule je přiložen obrázek 5.

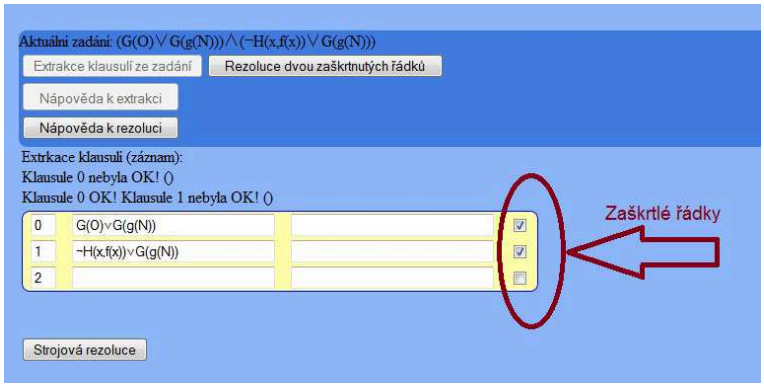


Obrázek 5: Schéma provedení exktrakce jedné klausule.

3.3.3 Schéma provádění kroku - rezoluce

Poznámka: pro pochopení schématu využijeme také obrázku 6.

- Program nejprve ověří, zda jsou zaškrtlé právě dva řádky (dvě klausule).
- Nad zaškrtlými řádky se provede operace rezoluce.
- Provede se porovnání výsledků (formule,vzniklá programem provedenou rezolucí vybraných řádků a formule, kterou uživatel zadal)



Obrázek 6: Schéma porvedení kroku rezoluce.

Poznámka: Tato podkapitola jen naznačuje, jak program provádí jeden krok rezoluce. Detaily jednotlivých kroků schématu (algoritmy apod.) jsou dále v textu vysvětleny.

3.4 Výběr zadání

Funkce pro generování (jedná se však o výběr) zadání (**generateTask(int obtiznost)**) je v programu umístěna ve třídě **PomocneFunkce2.cs**, která při vytváření instance požaduje dva parametry: label, do kterého se zadání zapíše a obtížnost vybraného zadání. Zadání se zapíše do příslušného labelu se stiskem tlačítka generace.

Před samotným klikem na tlačítko si může uživatel vybrat jednu ze tří možností obtížnosti zadání (Práce s programem je detailně popsána v kapitole 5):

1. lehká obtížnost,
2. střední obtížnost,
3. těžká obtížnost.

3.4.1 Výběr zadání - zjednodušený algoritmus

Poznámka: Před čtením samotného algoritmu pro výběr zadání je dobré se podívat na obrázek 7, ze kterého zjistíme strukturu souboru se zadáními.

1. Z prvního řádku souboru se zadáními se přečte číselná hodnota, která slouží programu jako informace o tom, kolik zadání se v souboru nalézá.
2. Program si spočítá, kolik zadání přísluší na jednu obtížnost podle vzorce:
   - **početZadáníNaObtížnost = kolikJeZadání / početObtížností;** Hodnota **počtu obtížností** je rovna konstantě 3 (lehká, střední, těžká obtížnost). Hodnota proměnné **kolikJeZadání** je rovna hodnotě prvního řádku souboru se zadáními.
3. Progam spočítá na základě vybrané obtížnosti a počtu zadání na jednu obtížnost **rozmezí** (minimum a maximum) pro pořadí zadání. Z tohoto rozmezí se pak bude vybírat **pořadové číslo zadání** ze souboru (například pro rozmezí 5 a 10 se může vybrat číslo 6 a pak se tedy vybere šesté zadání v souboru). Program určuje rozmezí následovně:
   - Je-li vybrána obtížnost „lehká“:
     - minimum = 0,
     - maximum = (počet zadání na obtížnost) - 1.
   - Je-li vybrána obtížnost „střední“:
     - minimum = počet zadání na obtížnost,
     - maximum = (počet zadání na obtížnost * 2) - 1.
   - Je-li vybrána obtížnost „těžká“:
     - minimum = (počet zadání na obtížnost * 2),
     - maximum = počet zadání celkem - 1.
4. Program náhodně vybere nějaké číslo „*x*“ v rozmezí minima a maxima.
5. Program ze seznamu všech zadání vybere x-té zadání.

Poznámka: na obrázku 7 je znázorněna ukázka souboru se zadáními. Za povšimnutí stojí, že formule obsahují vnitřní znaky programu, nikoli speciální znaky (viz. kapitola 3.1).

```
8
Vx[A(x)?(((A(x) & B(x)) & C(x)) & D(x))]
Vx[A(O) | B(x)] ? Vy[A(N)&B(y)]
Vx[A(x)|B(x)]?(!B(O)|B(O))
Vx(((A(x)|B(x))&A(x)) ? B(x))
[Vx[A(x)?(B(x) & VxC(f(x)))] & Ex!C(x)] ? (!A(x)&(B(y)|C(z)))
[Vx[A(x)?(B(x) & C(f(x)))] & Ex!C(x)] ? !(A(x)~B(x))
VxVxA(x)?B(x)
([Vx[A(x)? B(x)] & Ex!C(x)] ? Ex!A(x))
```

Obrázek 7: Soubor se zadáními.

3.5 Úpravy textové reprezentace formule před syntaktickou analýzou

Před procesem syntaktického ověření správnosti formule je vždy zapotřebí upravit vstup uživatele (nebo zadání) následujícím způsobem:

1. Vymazání všech mezer ve formuli.
2. Nahrazení všech typů závorek na závorky typu „(“ a „)“.

Pokud bychom tuto úpravu neprovedli, byla by gramatika syntaktického analyzátoru značně složitější (ve všech pravidlech by se muselo počítat s mezerami apod.) a tím by byl složitější i samotný proces syntaktické analýzy.

3.6 Syntaktické ověření správnosti formule

Tak jako všechny programy pracující s uživatelskými vstupy se i náš program musí vypořádat s nápaditostí uživatele. K syntaktickému ověření formule slouží třída **SyntaktickyAnalyzator.cs**, které se předá řetězec reprezentující formuli (řetězec už je v době předání upraven způsobem popsaným v kapitole 3.5.

Předaný řetězec se pak analyzuje gramatikou typu **LL1** (gramatika, u které je vždy pomocí jednoho znaku přesně dáno, které pravidlo gramatiky se použije). Syntaktická analýza je buď úspěšná nebo neúspěšná.

V případě úspěšné syntaktické analýzy program pracuje ve své činnosti dále a v případě neúspěšné syntaktické analýzy program také pokračuje v činnosti dále, ale navíc vypisuje uživateli informace o syntaktické chybě.

3.6.1 Implementovaná LL1 gramatika

Gramatika, kterou program využívá k ověření syntaktické správnosti formule se skládá z následujících pravidel (nutno podotknout, že gramatika pracuje s vnitřními znaky, nikoli se speciálními znaky):

1. S $\rightarrow$ F1F2
2. F1 $\rightarrow$ Pr(Arg) | V VaF1 | E VaF1 | !F1 | (F1F2)
3. F2 $\rightarrow$ |F3 | &F3 | ?F3 | ˜F3 | epsilon (První symbol „|“ tohoto řádku je vnitřním znakem disjunkce)
4. F3 $\rightarrow$ (F1F2) | F4
5. F4 $\rightarrow$ Pr(Arg) | V VaF4 | E VaF4 | !F4 | (F1F2)
6. Pr $\rightarrow$ ACi | BCi | ... | MCi (vyjma ECi, protože E je rezervovaný symbol)
7. Ci $\rightarrow$ 0Ci | 1Ci | ... | 9Ci
8. Arg $\rightarrow$ VaArg2 | Fu(Arg) | KoArg2

9. Arg2 → epsilon | ,Arg
10. Va → nCi | oCi | ... | zCi
11. Fu → aCi | bCi | ... | mCi
12. Ko → NCi | OCi | ... | ZCi (vyjma VCi, protože V je rezervovaný symbol)

Vysvětlení pravidel gramatiky je následující:

1. S je startovním neterminálem.
2. F1, F2, F3, F4 jsou neterminální symboly, kterými se analyzuje správné uzávorkování formule, správnost spojek apod.
3. Pr je neterminální symbol, který slouží k analýze predikátu.
4. Ci je neterminální symbol, který slouží k analýze jakékoliv číselné řady (indexy konstant, funkcí apod.).
5. Arg je neterminál, který slouží k analýze proměnné, funkce, konstanty, sekvence proměnných, sekvence funkcí, sekvence konstant.
6. Arg2 je neterminál, který slouží jako pomocný neterminál při analýze výše zmíněných sekvencí proměnných, funkcí a konstant.
7. Va je neterminál, který slouží k analýze jedné proměnné. Proměnná se skládá z jednoho z terminálních symbolů proměnných a libovolného číselného indexu (může být i prázdný symbol).
8. Fu je neterminál, který slouží k analýze jedné funkce. Funkce se skládá z jednoho z terminálních symbolů funkcí, indexu (může být i prázdný symbol) terminálního symbolu otevírací závorky, argumentů (argumentem může být proměnná, sekvence proměnných, funkce, sekvence funkcí, ...) a terminálního symbolu uzavírací závorky.
9. Ko je neterminální symbol, který slouží k analýze jedné konstanty. Konstanta se skládá z jednoho z terminálních symbolů konstant a libovolného číselného indexu (může být i prázdný symbol).
10. ABCDFGHIJKLM jsou terninální symboly určené pro predikáty.
11. NOPQRSTUWXYZ jsou terminální symboly určené pro konstanty.
12. abcdefghijklm jsou terminální symboly určené pro funkce.
13. nopqrstuvwxyz jsou terminální symboly určené pro proměnné.
14. V - je terminální symbol pro všeobecný kvantifikátor.

15. E - je terminální symbol pro existenční kvantifikátor.

Takto navržená gramatika umožňuje analyzovat formule, které mohou obsahovat nekonečně mnoho různých proměnných, funkcí, konstant a dokonce i predikátů (ačkoliv se značení predikátu s indexem moc nepoužívá).

Vlastní limita počtu proměnných (konstant, predikátů apod.)teoreticky neexistuje (existuje pouze nevlastní limita, která je rovna nekonečnu), ale v praxi je program limitován pamětí.

3.7 Převody formulí na různé reprezentace

Programu se ne vždy hodí mít formuli reprezentovanou řetězcem. V následujících podkapitolách jsou uvedeny funkce a jejich algoritmy, které umožňují převody formule z jedné reprezentace na nějakou jinou reprezentaci.

3.7.1 Proces převodu formule z řetězce do stromové struktury

Funkce převodu formule na stromovou strukturu je v celém programu jedna z nejdůležitějších funkcí. Umožňuje mnohem jednodušší způsob provádění úprav, které je nutno provézt, abychom převedli formuli do klausulární formy.

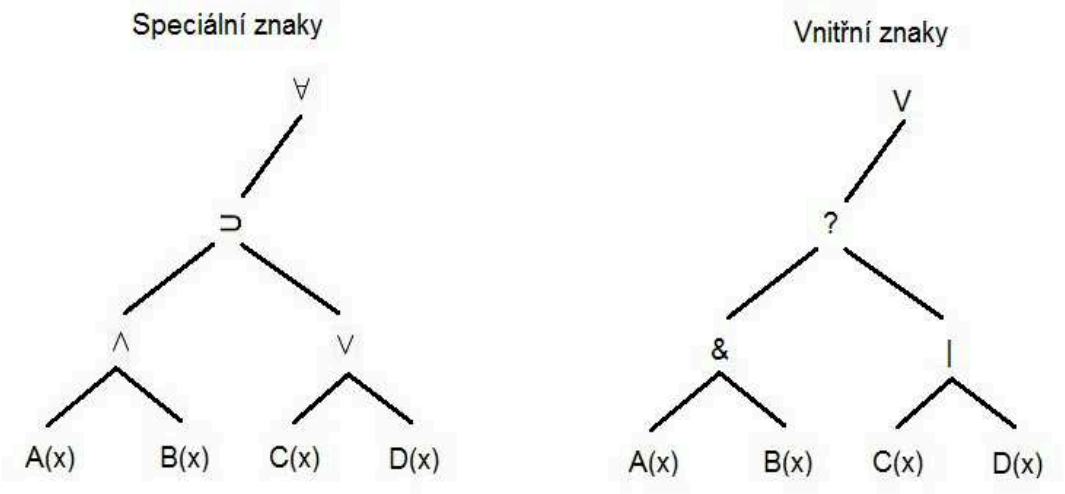
Funkce převodu je realizovaná ve třídě **VytvoreniStromu.cs**, které se při vytváření instance předá stringový parametr reprezentující formuli (Neboli aktuální zadání. Aktuální zadání nesmí obsahovat speciální znaky. Speciální znaky se nahrazují vnitřními znaky vždy před vytvářením instance třídy VytvoreniStromu.cs.). Samotný proces je rozdělen do několika kroků:

- Odstranění veškerých bílých znaků (mezery) a standardizace všech typů závorek na typ „(“ „)“.
- Nahrazení všech závorek stojících za predikáty - závorka stojící za predikátem se nahradí speciálním znakem a vygenerovaným identifikátorem (číslo). Důvodem je zjednodušení dalšího postupu a garance správného rozpoznání závorky reprezentující argumenty predikátu od závorky reprezentující část klausule.
- Nahrazení závorek reprezentujících části formule - proces, kdy se jednotlivé závorky (závorka v tuto chvíli už nemůže být nic jiného, než část formule) ve formuli postupně nahrazují speciálními sekvencemi znaků. Sekvence je tvořena jedním speciálním znakem, který je následován čísleným identifikátorem). Nahrazení závorek probíhá tak dlouho, dokud jsou ve formuli nějaké závorky.
- Vytvoření stromové struktury - proces, kdy se analyzuje aktuální formule (aktuální formule vždy obsahuje jednu z možností):
  - binární spojka (konjunkce, disjunkce apod.),
  - unární spojka negace,
  - predikát,

– kvantifikátor,
– speciální znak.

V prvním případě se binární spojka uloží do stromu a operace vytvoření stromové struktury se rekurzivně zavolá na řetězec před spojkou a pak také na řetězec za spojkou. Ve druhém, třetím i čtvrtém případě se do stromu uloží daná část formule a operace vytvoření stromové struktury se zavolá rekurzivně na část řetězce stojícího za danou částí formule (negace, predikát apod.). V pátém případě indikace speciálního znaku říká informaci, že je nejprve nutno nahradit speciální znak s identifikátorem za nějakou podformuli, což se okamžitě provede, a pak se proces vytváření stromu na tuto podformuli opakuje.

Názorná ukázka toho, jak může vypadat stromová reprezentace určité formule je znázorněna na obrázku 8.



Obrázek 8: Ukázka stromové reprezentace formule.

Důležitým faktem je, že výsledná stromová struktura umožňuje pohyb nejen od kořene k listům, ale umožňuje se stejným způsobem pohybovat od listů (resp. kteréhokoli uzlu stromu) směrem ke kořeni.

Takováto struktura je vhodná pro řadu úprav spojených s převodem klausule do klausulární formy. Krom své obrovské **výhody** (v pohybu v obou směrech - nahoru i dolu) však v sobě zkrývá jednu **nevýhodu** a tou je mnohem náročnější udržování platných vazeb mezi uzly.

Tato nepříjemná skutečnost v některých operacích způsobuje zmatek a je mnohem jednodušší neudržovat zpětné vazby (od uzlů směrem k jejich předchůdcům) a soustředit se na platnost vazeb směrem od uzl k jeho následníkům.

Takovýto postup bude korektní, protože úprava formule začíná vždy tím, že se převede stringová formule do stromu a teprve se stromem se pak pracuje. Na konci operace je strom převeden zpátky na stringovou formuli a výsledky jsou předloženy uživateli. Při převodu stromu na string se pak vůbec nepracuje se zpětnými odkazy (od uzlu k předchůdci) a z tohoto faktu pramení možnost neudržovat zpětné vazby mezi uzly (pokud je daná operace samozřejmě nepoužívá).

Navrhnutá stromová struktura Navrhnutá stromová struktura splňuje následující pravidla:

1. Základním stavebním prvkem stromové struktury je uzel. Uzel je třída, která obsahuje:
   (a) proměnnou „hodnota“ typu string, ve které je uložena hodnota uzlu,
   (b) proměnnou „predchudce“ typu uzel, ve které je uložen odkaz na předcházející uzel,
   (c) proměnnou „levy“ typu uzel, ve které je uložen odkaz na levý následující uzel,
   (d) proměnnou „pravy“ typu uzel, ve které je uložen odkaz na pravý následující uzel,
   (e) funkci „Clone()“, která slouží k vytváření takzvaných hlubokých kopií objektu. (Hluboká kopie se používá pro klonování (což je více méně kopírování) objektů, které jsou typu reference. Program s objekty typu reference pracuje zejména v upravách formule, které slouží k převodu formule do klausulární formy.)
2. Uzel, který má oba následníky prázdné (null), je listem.
3. V listu se nemůže objevit nic jiného, než predikát.
4. Unární spojka negace, existenční i všeobecný kvantifikátor mají vždy pravého následníka prázdného (null)
5. Unární spojka negace, existenční i všeobecný kvantifikátor mají vždy levého následníka neprázdného.
6. Binární spojky (konjunkce, disjunkce, implikace a ekvivalence) mají vždy levého i pravého následníka neprázdného.
7. Kořen stromové struktury nemá předchůdce (proměnná predchudce je rovna null).

3.7.2 Převod formule ze stromové struktury na řetězec

Převod formule ze stromové struktury na řetězec je další klíčovou funkcí (je umístěná ve třídě **PomocneFunkce.cs**), která je potřebná v situacích, kdy se provede nějaká operace nad formulí a výsledná formule se má prezentovat zpět uživateli. Proces už není tak složitý jako v převodu v opačném směru, protože formule ve stromu už nemůže obsahovat chyby (syntaktické).

Algoritmus převodu pracuje na principu přesného vkládání řetězců do výsledného řetězce (hlavní řetězec je při zahájení převodu prázdný). Funkce pro samotný převod je však rekurzivní a proto má (funkce) parametr, který nese informaci o indexu ve výsledném řetězci, na který (index) se bude nový řetězec vkládat.

Pro unární spojky, jako je negace, (stejný význam má v podstatě kvantifikátor) se v jednom kroku rekurze do výsledného řetězce vloží znak negace následovaný otevírací a zavírací závorkou („¬()“). Index zápisu pro další krok rekurze pak je roven indexu, na kterém se nalézá uzavírací závorka (s vložením řetězce na daný index se zbylé znaky posunou o délku vloženého řetězce).

Pro binární spojky, jako je konjunkce apod. se v jednom kroku rekurze do výsledného řetězce vloží znaky závorky mezi kterými je znak dané binární spojky („(∧)“). Index zápisu pro další levou rekurzi je pak roven indexu, na kterém stojí daná binární spojka. Index zápisu pro další pravou rekurzi však není roven indexu uzavírací závorky, protože v době, kdy se bude volat pravá rekurze, už bude řetězec dávno změněný a bude mít jinou délku (bude větší). Proto se index pro pravou rekurzi musí spočítat jako původní index pravé závorky plus celková délka vkládáného řetězce před původní index (zde je nutno dávat extrémní pozor na otevírací a uzavírací závorky, které automaticky obalují vkládané hodnoty, aby byla zaručena správná struktura řetězce).

Výsledkem převodu pak je sice řetězec se správnou strukturou, ale poněkud uživatelsky *nepěkný*. Obsahuje totiž velké množství závorek, které pro lidský mozek představují psychickou bariéru (formule vypadá složitě). Proto se současně s touto funkcí (tam, kde jsou výstupy převodu předávány k vizualizaci uživateli) používá funkce, která je schopna přebytečné závorky detekovat a odstranit (viz. kapitola 3.7.3).

3.7.3 Proces odstranění přebytečných závorek z formule

Odstranění přebytečných závorek je z hlediska stroje nepodstatný proces, avšak pro uživatele zcela klíčový (jak již bylo řečeno dříve). Funkce, která proces realizuje, je umístěna ve třídě **UpravaFormule_nepotrebneZavorky.cs**.

Algoritmus odstaňování závorek není triviální, a proto nastíním jeho fungování:

- Prochází se řetězec a hledají se znaky „(“, před kterými nestojí predikáty nebo funkce.
- Pro každou nalezenou otevírací závorku se najde příslušná uzavírací závorka (proces hledání uzavírací závorky v této práci nepopisuji, protože si myslím, že je relativně jednoduchý).
- Jakmile máme obsah závorky, zjistíme si její mohutnost (buď je mohutnost rovna jedné nebo dvěma). Ke zjištění mohutnosti závorky v tomto případě slouží kombinace rafinovaných funkcí, které jsou využívány v procesu převodu stringové formule na stromovou reprezentaci (ostranění závorek za predikáty, ostranění závorek specifikujících podformule). (Úprava závorky jako celku může dopadnou například takto: #2&#3 nebo #5. Z prvního příkladu bychom vyvodili mohutnost rovnu dvěma a z druhého příkladu bychom vyvodili mohutnost rovnu jedné.)

- Pokud je mohutnost závorky rovna dvěma, pak se závorky odstranit nemohou. V opačném případě se závorky odstraní.

3.8 Interní kontrola programu na detekci závěru

Program byl navrhnut tak, aby uměl pracovat s fromulemi, které obsahují nejméně jednu premisu a závěr (v posledních chvílích vývoje, kdy už byl text práce napsán se však dodělala do programu možnost vypnutí kontroly detekce závěru). Kontrola formule, která ověří, zda formule závěr obsahuje, je v programu spouštěna po vygenerování (vybrání) zadání.

Tato kontrola, protože není příliš složitá, není implementována ve vlastní třídě.

3.8.1 Detekce závěru - princip

Pokud formule po vygenerování (vybrání ze seznamu) zadání musí obsahovat závěr, pak je jasné, že po převedení zadání ze stringové reprezentace na stromovou reprezentaci bude kořen stromové reprezentace obsahovat spojku implikace (pokud kořen implikaci neobsahuje a obsahuje nějaký kvantifikátor, může být znak implikace až za sekvencí kvantifikátorů). Kontrola tedy pracuje ve třech krocích:

1. Vytvoření stromové reprezentace formule.
2. Porovnání hodnoty kořene této stromové struktury s vnitřní hodnotou implikace („?“).(Pokud se v kořeni nalézá kvantifikátor, pak se budeme zanořovat ve stromové struktuře tak dlouho doleva, dokud se nedostaneme na jiný uzel, než je kvantifikátor. Hodnotu takto dosaženého uzlu pak porovnáme s vnitřní hodnotou implikace.)
3. Vyhodnocení:
   (a) Pokud jsou hodnoty stejné, formule závěr obsahuje a uživateli je zpřístupněna funkcionalita programu pro další pokračování v Herbrandově proceduře.
   (b) Pokud jsou hodnoty různé, je vypsáno uživateli hlášení, že daná formule neobsahuje závěr.

Poznámka: Uživatel může kdykoli generovat nové zadání dle zvolené obtížnosti.

3.9 Úprava formule - negace

Je první úpravou formule (zadání), kterou musí uživatel v procesu hledání sporu ve formuli pomocí obecné rezoluční metody provézt (jak je definováno v Herbrandově proceduře XXX).

Úprava provedení negace je v programu zastoupena třídou **NegaceFormule.cs**. Třídě se při vytváření instance předá vrchol stromové struktury, která reprezentuje aktuální formuli.

Princip samotné negace je velmi jednoduchý. Stačí vytvořit nový uzel, jehož hodnota bude rovna negaci a tento nový uzel vložit před kořen celé formule.

3.10 Pravidla pro převod formule do klausulární formy

V této podkapitole si uvedeme všechna pravidla, která slouží k úpravě formule do klausulární formy. U každého pravidla bude uveden algoritmus, dle kterého program pravidlo realizuje.

3.10.1 Úprava formule - vytvoření existenčního uzávěru

Úprava vytvoření existenčního uzávěru je realizovaná třídou **Pravidlo1.cs** (jméno třídy se odvíjí od standardního pořadí uplatňování pravidel na formuli až do té doby, než je formule v klausulární formě). Třídě se při vytváření instance předá vrchol stromové struktury, která reprezentuje aktuální formuli (aktuální zadání). Na tuto stromovou reprezentaci, která je vrcholem jednoznačně definována se uplatní pravidlo vytvoření existenčního uzávěru.

Vytvoření existenčního uzávěru - zjednodušený algoritmus

Tato úprava probíhá na stromové struktuře, nikoli na stringové reprezentaci formule (převod stringové formule na stromovou reprezentaci viz. kapitola o převodu formule 3.7.1).

1. Rekurzivně se prochází stromová struktura od kořene k listům (program prochází strom levou rekurzí - tedy nejprve se vždy zanořuje doleva, pak doprava). Ukázka toho, jak rekurze probíhá, je zobrazena na obrázku 9.

2. V každém kroku rekurze se kontroluje, zda uzel stomové struktury obsahuje záznam s predikátem a v případě pozitivního výsledku program začne uzel prohledávat.

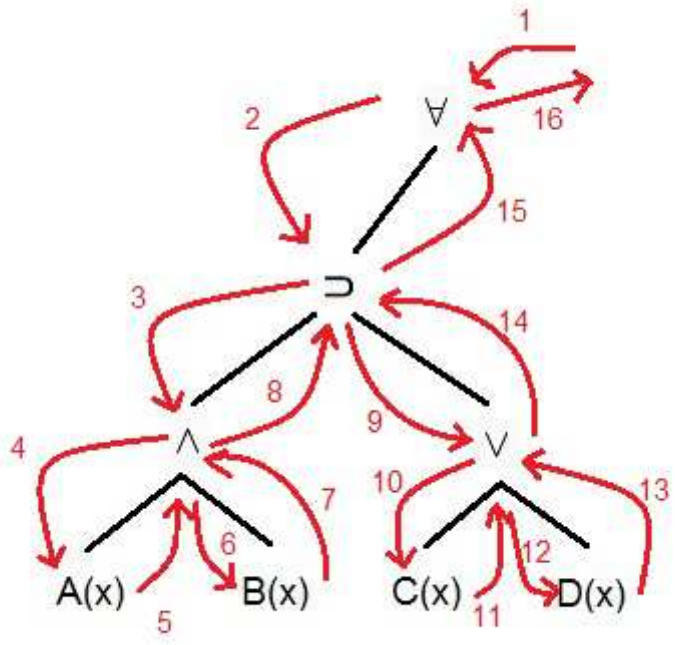
3. Pro každou nalezenou proměnnou v uzlu program zpětnou rekurzí od aktuálního uzlu směrem ke kořeni zjistí, zda se někde před aktuálním uzlem nachází takový uzel stromové reprezentace, který obsahuje kvantifikátor, který kvantifikuje hledanou proměnnou.

4. Pokud se najde takový uzel, který obsahuje kvantifikátor kvantifikující hledanou proměnnou, pak program pokračuje v rekurzi dále. Pokud se však zpětná rekurze dostane až do uzlu typu null, pak program detekuje, že pro hledanou proměnnou neexistuje kvantifikátor, který by ji kvantifikoval a vloží nový uzel typu existenční kvantifikátor kvantifikující hledanou proměnnou před kořen celé stromové reprezentace (nově vložený kvantifikátor se tedy stane novým kořenem celé stromové struktury).

Pro každou proměnnou se tak zajistí vazba nejméně k jednomu kvantifikátoru (všeobecnému či existenčnímu).

3.10.2 Úprava formule - eliminace nadbytečných kvantifikátorů

Úprava je zajištěna třídou **Pravidlo2.cs**. Třídě se při vytváření instance předá vrchol stromové struktury, která reprezentuje aktuální formuli (aktuální zadání).



Obrázek 9: Ukázka rekurzivního procházení stromové struktury.

Eliminace nadbytečných kvantifikátorů - zjednodušený algoritmus

Úprava pracuje s formulí, která je ve formě stromové reprezentace, nikoli se stringovou reprezentací.

Odstanění zbytečných kvantifikátorů probíhá ve dvou částech:

1. Formule se prochází rekurzivně směrem od kořene k listům. V každém kroku rekurze se kontroluje, zda aktuální uzel neobsahuje kvantifikátor a pokud ano, pak se od tohoto nalezeného kvantifikátoru program podívá do dosažitelných uzlů a hledá v nich proměnnou, která je kvantifikátorem kvantifikována. Programu stačí, aby našel jeden výskyt proměnné v kterémkoli listu a kvantifikátor je vyhodnocen jako potřebný - v opačném případě (když ani jeden list dosažitelný od nalezeného kvantifikátoru neobsahuje kvantifikovanou proměnnou) se kvantifikátor prohlásí za zbytečný a odstraní se.

2. Formule se prochází rekurzivně od kořene k listům. V každém kroku rekurze se kontroluje, zda aktuální uzel neobsahuje kvantifikátor a pokud ano, kontroluje se, zda hned za uzlem (v sekvenci kvantifikátorů) není uzel typu kvantifikátor kvantifikující stejnou proměnnou a v případě že ano, pak se uzel - aktuální uzel - ze stromu vymaže. V opačném případě aktuální uzel ve stromu zůstává.

Výsledkem operace eliminace nadbytečných kvantifikátorů je formule, která ke každé proměnné má právě jeden kvantifikátor.

3.10.3 Úprava formule - přejmenování proměnných

Úprava je zajištěna třídou **Pravidlo3.cs**. Třídě se při vytváření instance předá vrchol stromové struktury, která reprezentuje aktuální formuli (aktuální zadání) a také řetězec reprezentující aktuální formuli (aktuální zadání), aby nedocházelo ke zbytečnému zpomalování programu, protože, jak si řekneme dále, pro první část této úpravy je mnohem výhodnější pracovat s řetězcem, než se stromovou strukturou, zatímco ve druhé části je výhodnější pracovat se stromovou strukturou.

Přejmenování proměnných - zjednodušený algoritmus

Jak již bylo naznačeno výše, úprava se skládá ze dvou kroků:

1. Hledání všech proměnných z řetězce reprezentujícího aktuální zadání. Důvod je ten, že ve druhé fázi budeme hledat uzly kvantifikující stejné proměnné a bude nutno generovat nové názvy proměnných, které ještě nebyly nikde ve stromové reprezentaci použity.

2. Rekurzivní procházení stromové reprezentace od kořene k listům, kdy v každém kroku rekurze kontrolujeme, zda aktuální uzel obsahuje kvantifikátor. Pokud ano, pak v celém podstromu od tohoto kvantifikátoru až k listům hledáme všechny kvantifikátory a kontrolujeme, zda kvantifikují stejnou proměnnou a v případě, že ano, dochází k nahrazení hledané proměnné v celém podstromu specifikovaném nalezeným druhým kvantifikátorem (tedy od druhého kvantifikátoru až k listům). Generování nových názvů proměnných zajišťuje metoda, jejíž princip je následující:

   (a) Z původní proměnné si metoda nejprve zjistí první písmenko.

   (b) Ke zjištěnému písmenku začne postupně v cyklech přiřazovat indexy aritmetické řady An = 0,1,2,3, ... , n a pro každý název proměnné skládající se z písmene a indexu se zkontroluje, zda byl název někde ve stromové reprezentaci až doposud použit nebo ne. Cyklus přiřazování indexů končí tehdy, když se najde první název, který je pro danou aktuální formuli originální = dosud nepoužitý (nový název je pak přidán do bufferu všech proměnných, aby tato proměnná nebyla při dalším generování použita).

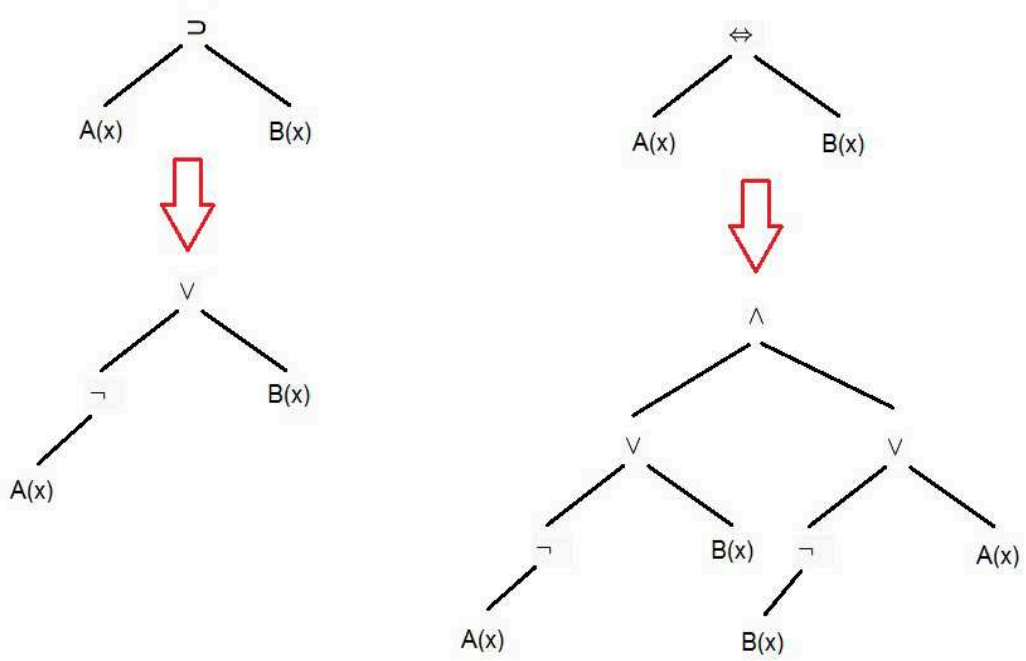
Výsledkem operace přejmenování proměnných je formule (ve stromové reprezentaci), která neobsahuje žádnou dvojici kvantifikátorů kvantifikující stejnou proměnnou.

3.10.4 Úprava formule - odstranění implikace a ekvivalence

Úpravu zajišťuje třída **Pravidlo4.cs**. Třída pracuje se stromovou strukturou formule a při vytváření instance se tedy třídě předá vrchol stromové struktury, která reprezentuje aktuální formuli (aktuální zadání).

Odstranění implikace a ekvivalence - zjednodušený algoritmus

1. Procházíme rekurzivně strom od kořene k listům a v každém kroku rekurze se kontroluje, zda aktuální uzel obsahuje binární spojku typu implikace a pokud ano, pak je tento uzel nahrazen dle obrázku 10.
2. Procházíme rekurzivně strom od kořene k listům a v každém zanoření se kontroluje, zda aktuální uzel obsahuje binární spojku typu ekvivalence a pokud ano, pak je tento uzel nahrazen dle obrázku 10.



Obrázek 10: Odstranění implikace a ekvivalence.

3.10.5 Úprava formule - přesun negace dovnitř

Úpravu zajišťuje třída **Pravidlo5.cs**. Třída pracuje opět se stromovou strukturou a při vytváření instance se jí předá vrchol stromové struktury, která reprezentuje aktuální formuli (aktuální zadání). Cílem úpravy je přesunout všechny negace co nejblíže k predikátovým symbolům. Po té, co jsou negace posunuty tak, že už se dále nedají posouvat (blíže k predikátům), se provádí takzvaná redukce negací - tedy dvě negace stojící ve stromové struktuře přímo za sebou se navzájem vyruší.

Přesun negace dovnitř - zjednodušený algoritmus Algoritmus je složený z cyklu, ve kterém se před iterací zjistí stav podmínky. Podmínka se ptá, zda je ve stromu někde

nějaká negace, za kterou se nenachází ani negace ani predikát. Pokud je tato otázka (podmínka) vyhodnocena jako pravda (to znamená, že někde ve stromu je tedy taková spojka negace, za kterou se nalézá například konjunkce), pak se zavolá operace negace, která prochází rekurzivně strom a pokud narazí na negaci, kterou jde posunout doprava (taková negace ve stromu musí někde být, protože byla nalezena podmínkou, avšak to nemusí být hned první nalezená negace, protože, jak víme, strom se prochází levou rekurzí.), dojde k posunutí negace blíže k predikátům (neguje se na základě pravidel uvedených v kapitole 3.14.1.

Vyrušení dvojí negace - zjednodušený algoritmus Strom se prochází rekurzivně od kořene k listům a v každém kroce rekurze se kontroluje, zda je aktuální uzel typu negace a pokud je, zkontroluje se, zda je i následník nalezené negace další negace a pokud ano, pak se obě negace vyruší.

3.10.6 Úprava formule - přesun kvantifikátorů doprava

Úpravu zajišťuje třída **Pravidlo6b.cs**. Třída pracuje opět se stromovou strukturou a při vytváření instance se jí předá vrchol stromové struktury, která reprezentuje aktuální formuli (aktuální zadání).

Cílem této úpravy je zmenšit pole působení kvantifikátorů na co nejmenší možné minimum - s ohledem na pravidla, která je nutno dodržet, aby byla upravená formule ekvivalentní.

Přesun kvantifikátorů doprava - zjednodušený algoritmus Algoritmus prochází rekurzivně stromovou strukturou a v každém kroku rekurze se kontroluje, zda je aktuální uzel typu kvantifikátor. V případě, že se najde kvantifikátor (označme jej kvantifikátorA), levou rekurzí (POZOR: V tomto odstavci je pojmem levá rekurze myšleno, že se rekurze zanořuje pouze doleva!) se zjistí, zda je za kvantifikátorem (někde) nějaká binární spojka a pokud je, pak se program z aktuálního nalezeného kvantifikátoru přesune na jeho nejposlednějšího přímého následníka (posouvání probíhá pouze na uzlech typu kvantifikátor) typu kvantifikátor (označme jej kvantifikátorB) a tohoto posledního následníka program zkouší distribuovat za nalezenou spojku, pokud je distribuce možná (pokud má distribuce smysl - tedy tehdy, pokud levý následník spojky proměnnou z kvantifikátoru obsahuje a pravý následník ne nebo naopak).

Pokud k distribuci dojde, původní uzel (v prvním kroce kvantifikátorB) je označen za neplatný a program se posune na předchůdce právě posunutého kvantifikátoru a zkouší v distribuci pokračovat.

Pokud nastane situace, kdy některý kvantifikátor nelze posunout směrem doprava, pak už se zbývající kvantifikátory (všechny ty kvantifikátory, které leží mezi kvantifikátoremA a KvantifikátoremB a jsou platné) ani posouvat nezkouší.

Po dokončení prvního rekurzivního procházení jsou ještě ve stromu uzly označené jako neplatné. Dalším rekurzivním průchodem stromu zajistíme vymazání těchto neplatných uzlů.

3.10.7 Úprava formule - skolemizace

Úpravu zajišťuje třída **Pravidlo7.cs**. Třída pracuje opět se stromovou strukturou a při vytváření instance se jí předá vrchol stromové struktury, která reprezentuje aktuální formuli (aktuální zadání). Cílem této úpravy je odstranění existenčních kvantifikátorů formule se změnou proměnných vázaných k těmto kvantifikátorům podle určitých pravidel - viz zjednodušený algoritmus.

Skolemizace - zjednodušený algoritmus Algoritmus skolemizace je rozdělen na dva kroky:

- Rekurzivní průchod stromovou reprezentací formule a uložení všech nalezených názvů konstant a funkcí. (Důvod je ten, že při odstraňování existenčního kvantifikátoru se musí proměnná, která je kvantifikátorem vázána, nahradit buď novou, nepoužitou funkcí nebo novou, nepoužitou konstantou. Čím se proměnná nahradí, záleží na okolnostech, které jsou popsány v dalším kroce.)

- Rekurzivní průchod stromem, kde se v každém zanoření kontroluje, zda aktuální uzel obsahuje existenční kvantifikátor a pokud ano, pak se do seznamu předchozích proměnných uloží všechny proměnné, které se vyskytují v přímých předchůdcích aktuálního uzlu. Přímí předchůdci aktuálního uzlu se hledají tak, že se z aktuálního uzlu zanořujeme do předchůdců tak dlouho, dokud nenarazíme na jiného předka, než je všeobecný kvantifikátor (př.: mějme formuli $\forall w \exists x \forall y \exists z$ A(w,x,y,z) - zde je přímým předkem kvantifikátoru $\exists z$ kvantifikátor $\forall y$). Pokud je počet předcházejících proměnných větší než nula, pak se všude ve stromové struktuře nahradí proměnná z aktuálního uzlu novou funkcí, která má jako své argumenty všechny předcházející proměnné. Pokud je počet předcházejících proměnných roven nule, pak se všude ve stromové struktuře nahradí proměnná z aktuálního uzlu novou konstantou. Nakonec se aktuální uzel vymaže.

3.10.8 Úprava formule - přesun kvantifikátorů doleva

Úpravu formule zajišťuje třída **Pravidlo8.cs**. Třída pracuje opět se stromovou strukturou a při vytváření instance se jí předá vrchol stromové struktury, která reprezentuje aktuální formuli (aktuální zadání).

Cílem této úpravy je přemístit všechny zbylé kvantifikátory ve formuli (v této chvíli jen všeobecné kvantifikátory) na začátek formule (jejich pořadí už nehraje roli).

Přesun kvantifikátorů doleva - zjednodušený algoritmus

- Přichystáme si prázdný buffer pro všechny nalezené uzly reprezentující kvantifikátory.

- Rekurzivně se prochází strom od kořene k listům a v každém zanoření se kontroluje, zda aktuální uzel je kvantifikátorem a když ano, pak se uzel označí za neplatný, vymaže se a jeho hodnota se uloží do připraveného bufferu.

- Všechny uzly z bufferu (všechny kvantifikátory) se vloží na začátek formule.

3.10.9 Úprava formule - použití distributivních zákonů

Úpravu formule zajišťuje třída **Pravidlo9.cs**. Třída pracuje opět se stromovou strukturou a při vytváření instance se jí předá vrchol stromové struktury, která reprezentuje aktuální formuli (aktuální zadání).

Cílem této úpravy je upravit formuli do tvaru konjunkce disjunkcí (tedy tak, aby se ve stromu reprezentujícího formuli neobjevila ani jedna spojka disjunkce nad spojkou konjunkce).

Po této úpravě (ne vždy je tato úprava nutná a z toho plyne, že někdy i po dřívější úpravě) se formule nachází v klausulárním stavu.

Použití distributivních zákonů - zjednodušený algoritmus

1. Program zjistí, zda se někde ve stromu formule objevuje taková disjunkce, na kterou lze (a má to smysl) uplatnit distributivní zákony. Distributivní zákony program uplatní v případě, kdy stromová reprezentace formule obsahuje disjunkci a následníci disjunkce jsou (jedna z možností):
   - konjunkce a negace,
   - konjunkce a predikát,
   - konjunkce a konjunkce,
   - konjunkce a disjunkce.
2. Uplatnění distributivního zákona na zjištěnou disjunukci.

3.11 Extrakce klausulí

Procedura extrakce klausulí je v programu zastoupena třídou **ExtrakceLiteralu.cs** a pracuje se stromovou strukturou formule, takže při vytváření instance se třídě předá vrchol stromové struktury, která reprezentuje aktuální formuli (aktuální zadání). Stromová struktura reprezentující zadání je již v klausulární formě.

Extrakce klausulí z formule, která je v klausulární formě není pro program nijak složitý proces. K extrakci je zapotřebí jednoho rekurzivního průchodu stromovou reprezentací formule.

3.11.1 Extrakce klausulí - zjednodušený algoritmus

Program prochází rekurzivně stromovou strukturu od kořene k listům a v každém kroku rekurze se zjišťuje, zda je v aktuálním uzlu binární spojka disjunkce a pokud ano, pak je jasné, že podstrom specifikovaný nalezeným vrcholem disjunkce specifikuje jednu klausuli.

Tato klausule se extrahuje a uloží do seznamu. K dalšímu zanoření za disjunkci nedojde, ale rekurze pokračuje dále do doby, než projde celý strom.

Výsledek extrakce klausulí je ještě upraven do přijatelnější podoby - tou je zřetězený seznam seznamů (list listů) řetězců (stringů). Důvodem převodu není nic jiného, než pohodlnější práce (například přístup přes indexy apod.).

3.12 Unifikace

Dalším krokem po extrakci klausulí je hledání sporu pomocí rezolučního pravidla aplikovaného vždy na nějaké dvě klausule. Proces, kdy se uplatňuje rezoluční pravidlo, však využívá jiný proces - Unifikaci.

Unifikace je zastoupena v programu třídou **Unifikace2.cs** jejíž instance při vytvoření požaduje dva argumenty typu string. Argumenty představují dva literály, pro které se bude hledat nejobecnější substituce.

Program využívá **Robinsnův unifikační proces pro hledání nejobecnější substituce** (2.3.7).

3.12.1 Unifikace dvou literálů - zjednodušený algoritmus v programu

- Zjištění informací o literálu 1:
  1. Přítomnost negace před literálem.
  2. Predikát v literálu.
  3. Zjištění mohutnosti argumentů predikátů (množství argumentů v závorce za predikátem)
- Zjištění informací o literálu 2 (identické s krokem 1).
- Testy před hledáním substituce. Pokud je kterýkoliv test vyhodnocen jako nepravda, pak hledání substituce končí (hledání substituce je nemožné nebo zbytečné).
  - Test na negace (v jednom literálu být negace musí a v druhém být nesmí).
  - Test na predikáty (v obou literálech musí být predikát stejný).
  - Test na mohutnosti závorek (obě mohutnosti musí být stejné).
- Hledání obecné substituce (Robinson).

Poznámka: Pod pojmem literál si můžeme představit sekvenci složenou maximálně z prvků:

1. 1x negace,
2. 1x predikát,
3. 1x závorka za predikátem. (Závorka však může být složena z libovolného množství čárkou oddělených argumentů. Argumenty mohou být v případě funkcí vnořené.)

3.13 Rezoluce

Je finálním krokem činnosti programu, ve kterém se snažíme hledat spor. Tento krok je v programu realizován třídami **Rezoluce.cs** (pro rezoluci nad dvěmi klausulemi) a **Rezoluce2.cs** (pro strojovou rezoluci provedenou programem od extrakce klausulí až po nalezení/nenalezení sporu).

Obě třídy mají jeden parametr - seznam seznamů (list listů) řetězců (stringů). Rozdíl mezi nimi je však ten, že třída Rezoluce.cs provádí jedno uplatnění rezolučního pravidla, zatímco třída Rezoluce2.cs provádí opětovné uplatňování rezolučního pravidla dokud najde/nenajde spor.

Poznámka: Proces, kdy mezi klausulemi hledáme spor může být „nekonečný", a proto byla po dohodě s Mgr. Markem Menšíkem, Ph.D. stanovena hranice maximálního možného uplatnění rezolučního pravidla na konkrétní konstantu - číslo **500**.

3.13.1 Strojová rezoluce - zjednodušený algoritmus

1. Program zavolá rekurzivní metodu *rezoluce(int radek1, int radek2)* na první dvě klausule (pokud je klausulí méně než dvě, rezoluce končí a spor se nenajde). V metodě rezoluce se provádí následující kroky:

   (a) Při každém volání metody rezoluce se inkrementuje počet provedených rezolucí (jakmile přesáhne počet provedených rezolucí hodnotu 500, proces hledání sporu končí s výsledkem nenalezení sporu).

   (b) Kontrola pozice (metoda rezoluce má dva parametry - int radek1, int radek2) první a druhé klausule. (Pozice určují pořadí jednotlivých klausulí, na které se aktuálně zkouší uplatnit rezoluce. Pokud se pozicí druhého řádku dostaneme za hranici počtu klausulí, zvýší se pořadí prvního řádku a pozice druhého řádku se nastaví na pozici rovnu pozice prvního řádku plus jedna. Pokud se dostaneme pozicí prvního řádku na pozici poslední klausule, pak rezoluce končí a k dalšímu rekurzivnímu volání sebe sama nedojde.)

   (c) Kontrola kombinace rezolvovaných řádků - program si uchovává v bufferu kombinace řádků, na které se rezoluce uplatnila(respektive zkoušela uplatnit). (Pokud se kombinace aktuálních řádků v bufferu objevuje, pak se rezoluce neprovádí.)

   (d) Unifikace klausulí. (Ve dvou cyklech se provádí postupná unifikace jednotlivých literálů klausulí a pokud se najde obecná substituce, uloží se tato obecná substituce do lokální proměnné (proměnná ve funkci rezoluce), pokud se obecná substituce nenajde, uloží se do zmíněné proměnné sekvence znaků upozorňující na nemožnost unifikace klausulí.)

   (e) Byla-li nalezena obecná substituce pro dvě klausule, substituce se pro dané klausule provede a pak se nad danými klausulemi provede rezoluce (Nejprve se ve dvou klausulích nahradí dle obecné substituce všechny termy, pak se opačné literály (ty literály, které jsou v obou formulích s opačnou negací) vymažou a pokud zbyde pouze prázdná klausule, dochází k ověření původní

délky obou klausulí - byly-li obě délky klausulí rovny jedné (1 literál), pak je spor dosažen korektně, v jiném případě nekorektně.)

(f) Nebyla-li nalezena obecná substituce k rezoluci nedochází.

(g) Metoda zavolá samu sebe s novými parametry řádků. (První řádek zůstává stejný a druhý řádek se zvýší o jedničku.)

2. Po dokončení rekurzivní metody se zkontroluje buffer všech klausulí na poslední pozici - pokud je na poslední pozici prázdná klausule (# neboli HASH), pak rezoluce skončí s tím, že spor byl nalezen. (Pokud se poslední klausule nerovná prázdné klausuli, pak se zavolá rekurzivní metoda rezoluce znovu na první dva řádky. Vše se opakuje, pokud není překročena hranice pěti set rezolucí.)

Algoritmus pro třídu Rezoluce.cs je velice podobný. Rozdíl je pouze v tom, že se rezoluce zkusí provést pouze jednou nad předanými dvěmi klausulemi (Obě klausule jsou předány prostřednictvím jednoho seznamu seznamů stringů. A ověření, že daný seznam obsahuje právě dvě klausule se děje před vytvářením instance třídy Rezoluce.cs. Algoritmus třídy Rezoluce.cs tedy nebudu dále rozvádět).

3.14 Kontrola uživatelských vstupů při úpravách formule do klausulární formy

Kontrolu uživatelských akcí zajišťuje třída **Kontrola.cs**. Úkolem třídy je zkontrolovat akci a uživatelský vstup s formulí vůči úpravě provedené programem.

Zjednodušený princip kontroly byl uveden v kapitole 3.3.1 a nyní se na proces kontroly podíváme detailněji z hlediska jednotlivých úprav formule do klasulární formy, z hlediska extrahování klausulí a z hlediska rezolvování klausulí.

3.14.1 Kontrola při kroku negace formule

Prvním krokem při Herbrandově proceduře je negování formule. Zde nastává velice nepříjemný problém spojený právě s negací formule, protože způsobů, jak zapsat negovanou formuli je nekonečně mnoho (Pro jednoduchost si představme pod symbolem A jakoukoli formuli. Zápisy A, ¬¬A, ¬¬¬¬A, ... , jsou ekvivalentní, ale jsou zapsány jinak.). Proces ověření, zda uživatel negoval formuli správně, s tímto problémem musí alespoň částečně počítat.

Algoritmus, který ověřuje, zda uživatel správně negoval formuli ze zadání je složen z následujících kroků:

1. Načtení a úprava (převod speciálních znaků, odstranění mezer, standardizace závorek) formule, kterou zadal uživatel.

2. Syntaktické ověření správnosti formule zadané uživatelem.

3. Převod upravené formule, kterou zadal uživatel ze stringové reprezentace do stromové reprezentace.

4. Úprava upravené formule, kterou zadal uživatel, kde se zajistí odstranění zbytečných závorek.

5. Načtení a uprava (převod speciálních znaků, odstranění mezer, standardizace závorek) aktuálního zadání.

6. Převod upraveného aktuálního zadání ze stringové reprezentace do stromové reprezentace.

7. Předání již negované formule ze zadání (ve stromové reprezentaci) třídě **GenerovaniVsechNegaci.cs**. Této třídě se při vytváření instance předává vrchol stromové struktury, která reprezentuje formuli, pro kterou se mají vygenerovat všechny ekvivalentní formule (ale podle omezených pravidel, která zajistí konečnost generování, protože jinak bychom generovali nekonečně dlouho). Třída po vytvoření instance zajistí vygenerování všech ekvivalentních formulí (Dle pravidel, které zajistí konečnost množiny všech ekvivalentních formulí. Ve formátu seznamu řetězců a seznamu stromových struktur. Poznámka: ve formátu seznamu řetězců jsou formule vždy zapsány bez zbytečných závorek.) vůči předané negované formuli. Pravidla, na základě kterých se generují všechny ekvivalentní formule vůči předané formuli, jsou:

   - $\neg(\ A\ ) \rightarrow \neg A$
   - $\neg(\neg A) \rightarrow \neg\neg A$, nebo A
   - $\neg\forall \rightarrow \exists\neg$
   - $\neg\exists \rightarrow \forall\neg$
   - $\neg(\ A \vee B) \rightarrow \neg A \wedge \neg B$
   - $\neg(\ A \wedge B) \rightarrow \neg A \vee \neg B$
   - $\neg(\ A \supset B) \rightarrow A \wedge \neg B$
   - $\neg(\ A \equiv B) \rightarrow (\neg A \vee B) \wedge (\neg B \vee A)$,kde si za A můžeme pro jednoduchost představit například predikát A(x) nebo formuli $\forall$A(x) apod. a za symbol $\rightarrow$ si můžeme představit proces, kdy program provede úpravu levé strany na stranu pravou (jednoduše řečeno si symbol šipky můžeme nahradit textem „může program přepsat na“).

8. Kontrola, zda je upravená formule (formule s vnitřními znaky, bez mezer, se standardními závorkami () ...), kterou zadal uživatel v seznamu všech ekvivalentních formulí vůči strojem správně negovanému zadání.

9. Vyhodnocení:

   (a) je-li upravená formule, kterou zadal uživatel v seznamu všech ekvivalentních formulí vůči strojem negovanému zadání, pak je uživatelova formule vyhodnocena jako správná a program vypíše náležitá sdělení.

(b) není-li upravená formule, kterou zadal uživatel v seznamu všech ekvivalentních formulí vůči strojem negovanému zadání, provede program srovnání stromové struktury uživatelem zadané formule se všemi stromovými strukturami, které jsou ekvivalentní vůči strojem negovanému zadání (Proces srovnání dvou struktur je zachycen v následující kapitole 3.14.2). Výsledek každého porovnání si program zapamatuje v bufferu a na základě výsledků v bufferu program vybere takovou strukturu (ze seznamu všech struktur ekvivalentních formulí vůči strojem negovanému zadání), která se nejvíce podobá struktuře formule, kterou zadal uživatel. Z takto vybrané sturktury se pak počítá výsledné hodnocení celého kroku a také se z ní vypočítává pozice první chyby ve formuli, zadané uživatelem.

3.14.2 Srovnání dvou stromových struktur

Srovnání dvou stromových struktur se děje v několika krocích a každý krok má na celkovém zhodnocení kroku uživatele podíl.

- Aby byl program alespoň částečně schopen hodnotit strukturu formule, prochází se najednou rekurzivně obě stromové struktury a v každém kroku rekurze se kontroluje aktuální uzel jedné struktury s uzlem druhé sturktury. V průběhu tothoto kroku se zaznamenává počet stejných uzlů (Uzly jsou stejné tehdy, když mají stejné řetězce v proměnné „hodnota“).
- Nejsou-li uzly stejné, vypočítává se ještě procentuální shodnost dvou řetězcových hodnot v uzlech tak, že si program nejprve zjistí délku obou řetězců, zjistí počet stejných znaků řetězců (na stejných místech pochopitelně - jde také o strukturu) a nakonec určí shodnost jako jednoduchý podíl počtu stejných znaků a délky delšího řetězce.
- Následným krokem je zjištění počtu uzlů stromových struktur.
- Celková shodnost dvou stromů se vypočítá jako: $shodnost = ((2 * pocetStejnychUzlu) \div pocetUzluCelkem) * 100$

Výsledkem srovnání dvou stromů je desetinné číslo x z intervalu s podmínkami $0 \leq$ x a zároveň x ≥ 1. (Když se číslo x vynásobí stem, dostaneme procentální shodnost.)

3.14.3 Vyhodnocení výsledků kontroly

Během procesu kontroly úprav, kterými se formule převádí do klausulární formy, se každý krok uživatele (krok = provedení jedné úpravy) vyhodnocuje a výsledky jsou vizualizovány. Informace, které program vizualizuje do tabulky umístěné na hlavní obrazovce programu (tabulka historie) jsou:

- strojová úprava (v tomto sloupci tabulky vidíme, jak by program provedl danou úpravu),

- co měl uživatel doplnit (v tomto sloupci tabulky je vizualizována formule, kterou měl dle odhadu programu uživatel doplnit),
- co uživatel doplnil (v tomto sloupci tabulky je uvedena formule, která podle uživatele měla po uplatnění daného pravidla vyjít),
- použité pravidlo (v tomto sloupci tabulky je slovně napsáno pravidlo, které se při úpravě uplatnilo),
- hodnocení kroku (v tomto sloupci tabulky je v rozmezí hodnot 1 až 5 hodnocena úprava jako celek).

Vysvětlení k položkám asi není třeba. Jen bych rád upozornil na užitečnou vlastnost bodu číslo 3. Ve sloupci tabulky „co uživatel doplnil" se v případě, že uživatel zadal syntakticky správnou formuli a vybral správně pravidlo, které se na zadání má v danou chvíli uplatnit, vypíše formule, kterou uživatel zadal s vizualizací hranice správné a chybné části zadané formule. Oblast, ve které je formule správná, je podbarvena zelenou barvou, zatímco oblast formule, která je špatná, je podbarvena barvou červenou.

Dále bych chtěl upozornit na způsob, jakým je úprava jako celek zhodnocena.

Pokud uživatel zadá správnou úpravu (pokud nezadá správnou úpravu, je hodnocen známkou 5) a také správnou formuli, je hodnocen známkou 1. Pokud uživatel zadá správnou úpravu a syntakticky špatnou formuli, je hodnocen pětkou. Pokud však uživatel zadá správnou úpravu a syntakticky správnou formuli, dochází k porovnání formule, kterou zadal uživatel s formulí, která vznikla strojovou úpravou a zde může být uživatel hodnocen v rozmezí 2 - 4. Vzorec, dle kterého se spočítá hodnocení, je roven:

- hodnocenikroku = (automatickeZhorseni + (2 - ((pozicePrvniNalezeneChyby / delkaF1) * 2))), kde automatickeZhorseni je rovno konstantě 2, pozice první nalezené chyby je rovna indexu prvního rozdílného písmenka v zadání a uživatelském vstupu a délkaF1 je délka delšího řetězce (délka zadání nebo uživatelského vstupu).

3.14.4 Kontrola správného vytvoření existenčního uzávěru

Kontrola, zda uživatel správně vytvořil existenční uzávěr, není složitý proces. Při kontrole existenčního uzávěru se objevuje pouze jeden problém a to, že uživatel nemusí zadat kvantifikátory existenčního uzávěru v pořadí, v jakém je přidá program. Pomocí metody na srovnávání dvou stromů tedy nemůžeme tuto kontrolu provézt.

Problém určení, zda uživatel vytvořil exist. uzávěr správně, je v programu řešen následovně (V postupu vynechám podrobnosti, jak se ze zadání a uživatelského vstupu došlo ke stromové reprezentaci formulí, protože tento postup už byl několikrát zmiňován. Tento postup bude také vynechán ve zbývající části celé diplomové práce.):

- Získáme stromovou reprezentaci uživatelského vstupu.
- Získáme stromovou reprezentaci zadání.

- Ze stromové reprezentace uživatelského vstupu získáme pomocí levé rekurze (od kořene se budeme zanořovat pouze do levého následníka a v každém kroku si budeme do seznamu stringů ukládat hodnoty uzlů a to tak dlouho, dokud bude zpracovávaný uzel typu existenčního kvantifikátoru) seznam stringových hodnot všech prvků existenčního uzávěru.
- Stejným způsobem získáme prvky existenčního uzávěru zadání.
- Setřídíme abecedně oba seznamy.
- Od stromové struktury uživatelského vstupu oddělíme existenční uzávěr. (K tomuto kroku použijeme třídu **ziskaniStromBezExistencnihoUzaveru.cs**, které při vytváření instance předáme vrchol stromové struktury, která reprezentuje uživatelem zadanou formuli.) Princip oddělení existenčního uzávěru je jednoduchý: procházíme strom od kořene k listům a dokud je aktuálně zpracovávaný uzel typu existenční kvantifikátor, tak uzel smažeme.
- Stejnou procedurou získáme zadání bez existenčního uzávěru.
- Stromovou strukturu uživatelského vstupu bez existenčního uzávěru převedeme na řetězec (převod stromové reprezentace na řetězec viz. kapitola 3.7.2) a odstraníme přebytečné závorky (viz kapitola 3.7.3).
- Stejnou proceduru provedeme pro stromovou strukturu zadání.
- Setřizený seznam řetězců (kde jsou uloženy prvky existenčního uzávěru uživatelského vstupu) uživatelského vustupu převedeme na jeden řetězec.
- Stejnou operaci provedeme se seznamem řetězců, reprezentující prvky existenčního uzávěru zadání.
- Sloučíme vzniklý řetězec reprezentující existenční uzávěr uživatelského vstupu se stringovou reprezentací stromové struktury uživatelského vstupu bez existenčního uzávěru.
- Stejnou operaci provedeme se seznamem řetězců pro zadání a stringovou reprezentací stromové struktury zadání bez existenčního uzávěru.
- Provedeme porovnání obou vzniklých sloučených řetězců a vyhodnocení kroku (viz kapitola 3.14.3).

3.14.5 Kontrola správné eliminace nadbytečných kvantifikátorů

Proces kontroly, zda uživatel správně eliminoval nadbytečné kvantifikátory, není složitý. Pravidlo samotného odstranění přebytečných kvantifikátorů je přesně dáno. Algoritmus této úpravy zní:

1. Program si ze stringové reprezentace zadání vytvoří stromovou strukturu, nad kterou provede operaci eliminace nadbytečných kvantifikátorů (viz. kapitola 3.10.2).

2. Výsledek operace se převede na stringovou reprezentaci a z výrazu se odstraní přebytečné závorky.

3. Uživatelem zadaná formule se standardizuje (odstranění mezer, standardizace závorek apod.) a odstraní se z ní přebytečné závorky.

4. Program provede porovnání obou formulí a vyhodnocení celého kroku (viz. kapitola 3.14.3).

3.14.6 Kontrola správného přejmenování proměnných

Kontrola správného přejmenování je proces, který v sobě zkrývá jeden závažný problém. Jedná se o to, že uživatel může při kroku přejmenování proměnných použít jiné názvy proměnných než program (který, jak víme z kapitoly 3.10.3, generuje nové názvy za pomocí stejného písmenka proměnné a rozdílného indexu).

Kontrola je tedy na tento problém řádně připravena a algritmus samotné kontroly vypdá následovně (hodně zjednodušeně):

1. Získáme stringovou reprezentaci zadání a odstraníme přebytečné závorky.

2. Získáme stringovou reprezentaci formule zadané uživatelem a odstraníme přebytečné závorky.

3. Vyextrahujeme ze zadání všechny proměnné do seznamu (v dalších krocích je tento seznam označen jako SEZNAM A). (K tomuto procesu slouží třída **UpravaFormuleVyjmutiPromennych.cs**, které se při vytváření instance předá stringová reprezentace formule, ze které se extrahují všechny proměnné do seznamu (listu) řetězců (stringů) a zároveň se provede označení všech proměnných v předané formuli prostřednictvím vložení znaků „@“ před i za proměnnou.)

4. Vyextrahujeme z uživatelem zadané formule všechny proměnné do seznamu (SEZNAM B, proces extrakce je stejný jako v předchozím kroce).

5. Provedeme srovnání formulí (ve kterých jsou v obou případech proměnné odděleny znaky „@“) pomocí třídy **SrovnaniFormuliSRuznymiPromennymi.cs**. Třída při vytváření instance požaduje předání všech extrahovaných proměnných z uživatelem zadané formule (SEZNAM A), všech extrahovaných proměnných ze zadání (SEZNAM B), formuli zadanou uživatelem s vyznačenými proměnnými a formuli ze zadání s vyznačenými proměnnými (třída má ještě jeden parametr, který je v tuto chvíli nepodstatný, protože se využívá při kontrole procesu skolemizace). Ve třídě SrovnaniFormuleSRuznymiPromennymi.cs se postupně prochází SEZNAM A. V každém i-tém kroku průchodu se zkusí vyhodnotit podmínka, zda je i-tá proměnná v SEZNAMU A stejná jako i-tá proměnná ze SEZNAMU B. Pokud ano, v SEZNAMU B se znepřístupní změna proměnných na všech pozicích, na kterých se vyskytuje i-tá proměnná seznamu A. Pokud i-tá proměnná SEZNAMU A není stejná jako i-tá proměnná SEZNAMU B, pak se zkontroluje zda je na i-té pozici v

SEZNAMU B povolená změna proměnné (pokud změna na i-té pozici povolena není, pak uživatel přejmenoval proměnné špatně a ze závěrečného kroku vyplyne, že uživatel udělal chybu) a pokud ano, dojde k přepisu i-té proměnné SEZNAMU B i-tou proměnnou SEZNAMU A a v celém SEZNAMU B se místo staré i-té hodnoty objeví nová i-tá hodnota SEZNAMU A (samozřejmě pouze na těch položkách, kde je povolena změna hodnoty). Pak se znemožní zápis na všech pozicích SEZNAMU B, kde došlo k nahrazení proměnné. Po dokončení průchodu se všechny proměnné ze zadání nahradí proměnnými ze SEZNAMU B (po řadě).Následným krokem je již ověření, zda se updatované zadání rovná uživatelské formuli (ze které jsou ještě vymazány speciální znaky oddělující proměnné). Pokud se zadání a uživatelská formule rovnají, uživatel uplatnil pravidlo přejmenování proměnných správně, v opačném případě špatně.

6. Vyhodnocení celého kroku (viz. kapitola 3.14.3).

3.14.7 Kontrola odstranění implikace a ekvivalence, přesunu negace doprava, přesunu kvantifikátorů doprava, uplatnění distributivních zákonů

Pro všechny úpravy uvedené v nadpisu této podkapitoly se používá stejný algoritmus ověřující správnost uživatelem zadané formule vůči výsledku úpravy. Algoritmus pro ověření správnosti uživ. formule vůči zadání je následující:

1. Na aktuální zadání se uplatní (některá) úprava z nadpisu kapitoly.

2. Výsledek, který po aplikování úpravy dostaneme, použijeme při vytváření instance třídy **GenerovaniVsechEkvFormuliZakladni.cs**. Tato třída při vytvoření instance vytvoří k předané formuli všechny ekvivalentní formule jen tím, že mění vnitřní strukturu stromu (prohazuje levé a pravé následníky v konjunkcích a disjunkcích). Algoritmus této metody neuvádím, protože je velmi těžké jej jednoznačně slovně vysvětlit (pro představu o jeho složitosti - algoritmus používá dohromady cykly a rekurzi).

3. Získáme formuli zadanou uživatelem (vymazání mezer, standardizace, ...) a odstraníme v ní přebytečné závorky.

4. Zkontrolujeme, zda se upravená formule zadaná uživatelem nachází v seznamu všech vygenerovaných ekvivalentních formulí vůči strojem dosaženému výsledku.

5. Vyhodnocení: pokud uživatel zadal formuli, která se nalézá ve všech vygenerovaných ekvivalentních formulích vůči výsledku, který byl dosažen danou strojovou úpravou, vyhodnocení probíhá standardně. Pokud však uživatel zadal formuli, která se v seznamu všech vygenerovaných ekvivalentních formulí nenachází, pak se použije v principu stejný postup jako v kapitole 3.14.1, kdy uživatel nezadal ani jednu negaci, která je v seznamu všech dosažitelných negací.

3.14.8 Kontrola skolemizace

V této kontrole se potkáváme s podobným problémem jako v kapitole 3.14.6. Pro ověření správnosti uživatelem zadané formule vůči výsledku strojové úpravy (skolemizace) se používá následující algoritmus:

1. Získáme formuli zadanou uživatelem (vymazání mezer, standardizace, ...) a odstraníme v ní přebytečné závorky.

2. Získáme z formule zadané uživatelem všechny použité konstanty, funkce a proměnné jako seznam řetězců. K získání takového seznamu použijeme třídu **UpravaFormuleVyjmutiPromennychFunkciKonstant.cs** (tato třída má v principu stejnou funkci jako třída UpravaFormuleVyjmutiPromennych.cs z kapitoly 3.14.6 s tím rozdílem, že je schopna extrahovat mimo proměnné také funkce a konstanty + v předané formuli vyznačuje kromě proměnných také funkce a konstanty).

3. Stejným způsobem získáme všechny použité konstanty, funkce a proměnné ze zadání.

4. Vytvoříme instanci třídy SrovnaniFormuliSRuznymiPromennymi.cs stejně jako v kapitole 3.14.6, pomocí které nebudeme zjišťovat, zda se prvky použité v SEZNAMU A (značení z kapitoly 3.14.6) dají převést na prvky SEZNAMU B jako v kapitole 3.14.6, ale převedeme co nejvíce hodnot ze SEZNAMU A do SEZNAMU B (stejným způsobem jako v kapitole 3.14.6). Tím, že budeme přenášet prvky SEZNAMU A do SEZNAMU B, dostaneme nejpřesnější možný seznam všech funkcí, konstant a proměnných vyskytujících se ve výsledku operace skolemizace.

5. Vygenerování všech ekvivalentních formulí k formuli, která vznikla strojem provedenou skolemizací nad zadáním. (Ještě před samotnou generací všech ekvivalentních formulí se v předávané formuli provede nahrazení všech funkcí, konstant a proměnných za funkce, proměnné a konstanty ze získaného seznamu v předchozím kroku.)

6. Zbylá část algoritmu - od kontroly, zda uživatel zadal jednu z možných správných formulí až po vyhodnocení - je stejná jako v předchozí kapitole 3.14.7 (od čtvrtého bodu algoritmu).

3.14.9 Kontrola přesunu kvantifikátorů doleva

V této kontrole se opět potýkáme s podobným problémem jako v kapitole 3.14.4 - problém spojený s tím, že nevíme, v jakém pořadí uživatel všeobecné kvantifikátory do formule napíše. Algoritmické řešení problému je více méně stejné jako v kapitole 3.14.4, s výjimkou, že v algortimu nepracujeme s uzly typu existenční kvantifikátor, ale pracujeme s uzly typu všeobecný kvantifikátor (existenční kvantifikátory v době kontroly ve formuli ani neexistují).

3.15 Kontrola uživatelských vstupů při extrakci klausulí ze zadání

Kontrola klausulí v procesu extrahování klausulí je oproti předchozím kontrolám značně jednodušší. Kontrola extrakce klausulí není implementována ani ve své vlastní třídě. Při extrakci klausulí program předpokládá, že uživatel bude klausule ze zadání extrahovat stejným systémem, jako když se čte věta - tedy zleva doprava (první extrahuje klausuli úplně vlevo, pak druhou zleva atd.). A díky tomuto předpokladu programu stačí, že si drží všechny klausule v paměti a při každém kroku extrakce se pomocí jednoduchého porovnání stringů zjistí, zda uživatel provedl extrakci správně nebo špatně (uživatelský vstup je vždy standardizován z hlediska mezer, závorek). Pořadí jednotlivých literálů klausule je taky pevně dáno (jako u pořadí klausulí).

3.16 Kontrola uživatelského vstupu při rezoluci dvou označených řádků

Finálním krokem Herbrandovy procedury je hledání sporu pomocí uplatňování rezolučního pravidla na vždy dvě klausule. V tomto procesu už má uživatel k dispozici sadu klausulí a u každé z nich je vždy zaškrtávací políčko. Aby došlo k procesu uplatnění rezolučního pravidla na určité dvě klausule, musí uživatel vždy pomocí zaškrtávátka vybrat právě dvě klausule. Pokud nevybere právě dvě, pak se rezoluce nespustí a dojde pouze k znovuobnovení původních klausulí.

Předtím, než uživatel spustí rezoluci nad dvěmi klausulemi by měl do posledního řádku (který je vždy prázdný právě pro zapsání klausule) zapsat klausuli, která podle něj vznikne po uplatnění rezolučního pravidla na ním vybrané dvě klausule. (Symbol prázdné klausule je v programu reprezentován znakem # .)

Algoritmus samotné kontroly je následující:

1. Nejprve se zkontroluje, zda byly zaškrtlé právě dva řádky (dvě klausule).
2. Z obou zašrktlých řádků se vyextrahují klausule.
3. V klausulích se nahradí speciální znaky vnitřními znaky (viz. 3.1), provede se standardizace závorek a mezer.
4. Klausule jsou zatím v podobě řetězců - převedou se tedy do jednoho zřetězeného seznamu seznamů (list listů) řetězců (stringů). Celá jedna klausule je uložena do jednoho listu řetězců (kde jednotlivé řetězce odpovídají jednotlivým literálům). Obě klausule jsou zřetězeny proto, protože s takovou strukturou pracují obě třídy Rezoluce.cs i Rezoluce2.cs (které rezoluční pravidlo implementují), pomocí kterých dochází k fyzické rezoluci v programu.
5. Předání vytvořeného zřetězeného seznamu seznamů třídě Rezoluce.cs (princip fungování viz. kapitola 3.13).
6. Upravení vstupu uživatele - standardizace závorek, mezer.
7. Upravení vstupu uživatele ze znaků vnitřní reprezentace na znaky speciální (viz. 3.1). Důvod je ten, že výsledek rezoluce už chceme uživateli vizualizovat ve formátu upravených znaků na znaky speciální.

8. Porovnání výsledku rezoluce (kde jsou ještě upraveny znaky vnitřní na znaky speciální) a uživatelského vstupu. Pokud jsou formule stejné, pak program pokračuje vizualizací. Pokud však jsou formule rozdílné, ještě dochází ke kontrole, zda uživatel pouze neprohodil jednotlivé literály.
9. Vizualizace výsledků.

Vizualizace výsledků rezoluce je velice jednoduchou záležitostí. V případě správné rezoluce se přidá do tabulky všech klausulí nový řádek, ve kterém je pořadové číslo řádku, výsledek operace rezoluce a použitá substituce. Celý řádek je také zabolokován proti přepsání z bezpečnostních důvodů (aby se program nezhroutil apod.). V případě, že je rezoluce nesprávná (buď je nekorektní, nebo nemá význam rezolvovat dané řádky), se nový řádek do tabulky všech klausulí nepřidá. Místo toho se uživateli zobrazí zpráva, že jím zatržené řádky rezolvovat nelze.

3.17 Nápověda

Systém nápovědy v celém programu je rozdělen do čtyřech částí:

1. Nápověda v procesu převádění klausule do klausulární formy.
2. Nápověda při procesu extrahování klausulí ze zadání.
3. Nápověda při provádění rezoluce vybraných řádků.
4. Nápověda, která uživateli ukáže celý proces hledání sporu mezi jednotlivými klausulemi.

3.17.1 Nápověda v procesu převádění klausule do klausulární formy

V procesu, kdy uživatel převádí pomocí programu klausuli do klausulární formy, mu je k dispozici nápověda, jejíž princip fungování je velice jednoduchý (né algoritmicky, ale myšlenkou).

Když chce uživatel zobrazit nápovědu, klikne na tlačítko „Nápověda“ a nápověda okamžitě začne zpracovávat aktuální zadání a po řadě na zadání zkouší uplatňovat sadu desíti pravidel (negace, vytvoření existenčního uzávěru atd.) a po každém uplatnění zkontroluje řetězcovou strukturu formule před úpravou a po úpravě a pokud jsou řetězcové struktury formule jiné, pak nápověda zahlásí naposledy uplatněné pravidlo uživateli. Nikoliv však v přímé formě jako například „proveďte pravidlo 4“, ale spíše ve formě věty, která většinou uživateli řekne, proč má danou úpravou uplatnit. Nápověda může například napovědět: „Ve formuli se vyskytují na různých místech stejné proměnné, zkuste je přejmenovat.“

3.17.2 Nápověda v procesu extrakce klausulí

Při situacích, kdy uživatel extrahuje klausule ze zadání, je nápověda hodně statická a napovídá pouze větu, že jednotlivé klausule jsou v zadání odděleny symbolem konjunkce. Statická je z důvodu, že samotný proces extrakce je velice jednoduchý.

3.17.3 Nápověda při rezolvovaní klausulí

Tak jako nápověda v procesu úpravy formule do klausulární formy je i nápověda v procesu rezolvování klausulí jednoduchá, co se myšlenky týče.

Zjednodušený algoritmus nápovědy je následující (algoritmus je zjednodušený proto, že všechny principy, které používá, byly už několikrát popsány):

Nápověda si nejprve načte všechny klausule, které má uživatel současně k dispozici a tyto klausule si převede do zřetězeného seznamu seznamů řetězců. Ve dvou cyklech pak předává vždy dvě nejkratší klausule třídě Rezoluce a kontroluje, zda byla nalezena nějaká obecná substituce a v případě že ano (a klausule, která rezolucí vznikne není v seznamu aktuálních klausulí), napoví uživateli, že může rezolvovat dané klausule. Po napovězení ukončí nápověda svoji činnost.

4 Dokumentace

Pro vytvoření dokumentace k projektu byl použit nástroj doxygen. Jeho snadné použití a pěkné výstupy několikanásobně převyšují kvality nástrojů zabudovaných v programu Visual Studio 2008 (což byla vývojová platforma programu po celou dobu vývoje). Další nesmírnou výhodou nástroje doxygen je, že je volně dostupný ([2]).

Pro vytvoření dokumentace má nástroj doxygen v zásadě dva požadavky. Prvním je proces, kdy pomocí zabudovaného grafického průvodce nastavujeme parametry pro potřebnou dokumentaci (parametrů je opravdu hodně) a druhým, časově náročnějším, požadavkem je mít zdrojové texty programu řádně okomentovány. Ukázka, jak správně komentovat, je zobrazena na obrázku 11.

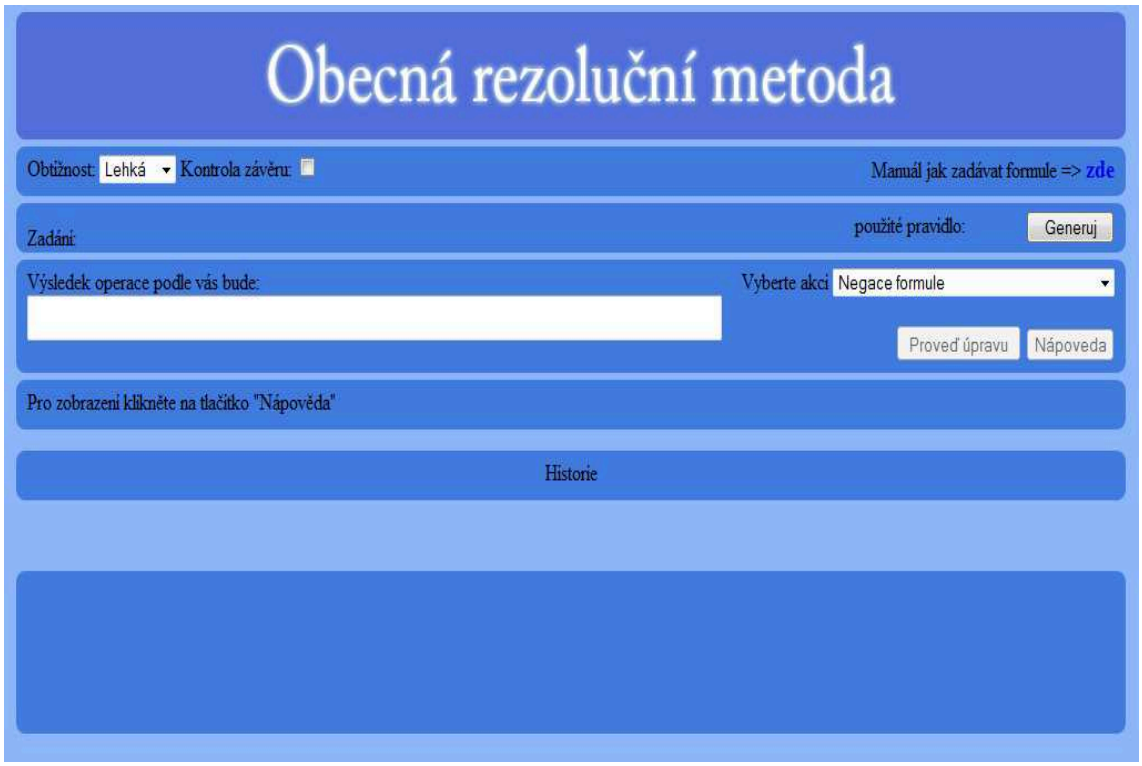
```
/// <summary>
/// Metoda, která ve formuli standardizuje závorky a odstraní z ní mezery.
/// </summary>
/// <param name="text">Vstupní formule (text).</param>
/// <returns>Upravená formule (text).</returns>
public string odstranMezeryAStandardizujZavorky(string text)
{
```

Obrázek 11: Jak správně dokumentovat zdrojové kódy.

Ačkoliv komentáře vypadají děsivě (sekvence lomítek apod.), lze si na ně poměrně rychle zvyknout. Generování struktury jednoho komentáře v programu Visual Studio 2008 a všech následníků i předchůdců, se kterými jsem se v praxi setkal, zabere necelou vteřinu. Protože při trojnásobném stisknutí znaku lomítka („/“) se celá struktura jednoho komentáře vygeneruje sama. Pak stačí vyplnit popis, návratové hodnoty apod.

5 Práce s programem

Program je vytvořen jako webová aplikace. Pro testovací účely jsem vytvořil dočasný web na adrese rezolucnimetoda.aspone.cz, kde je program umístěn. Práce s programem je intuitivní a pohodlná. Grafické uživatelské rozhraní programu je uvedeno na obrázku 12.



Obrázek 12: Grafické uživatelské rozhraní programu.

5.1 Generování zadání

Po načtení hlavní stránky stojí uživatel před prvním krokem a tím je vygenerování zadání. Pro vygenerování zadání si uživatel zvolí nejprve obtížnost zadání a pak zvolí, zda má program provézt kontrolu na detekci závěru ve formuli. Následně uživatel klikne na tlačítko generuj. Pokud uživatel v předchozím kroku zatrhl volbu, že chce provést kontrolu závěru zadání a zadání neobsahuje závěr, pak se za vygenerované zadání zapíše hlášení, že v zadání chybí závěr a uživateli nezbývá nic jiného, než aby si vygeneroval jiné zadání (v případě, že zadání obsahuje závěr může uživatel pokračovat například negováním formule).

5.2 Úprava formule do klausulární formy

Vygenerované zadání uživatel musí před rezolucí převést do klausulární formy. Proces převodu je rozdělen do deseti kroků (ne vždy je nutno provézt všech deset kroků). V každém kroku si uživatel musí prohlédnout zadání a pak vybrat akci (vybrat úpravu), která se má nad daným zadáním provézt. Následně uživatel zapíše do příslušné kolonky takovou formuli, která podle něj vznikne po aplikování pravidla, které vybral, na aktuální zadání. Po vybrání akce, zapsání formule, už stačí kliknout na tlačítko „proveď akci“.

Proces úpravy do klausulární formy je zaznamenáván v tabulce „historie“.

5.3 Extrahování klausulí

Po převodu do klausulární formy je uživateli umožněno extrahovat klausule ze zadání. Extrahování klausulí se děje postupně v krocích a v každém kroce má uživatel zadat do prázdného řádku tabulky klausuli, která se podle něj bude extrahovat (klausule program extrahuje ve směru zleva doprava). Pak už stačí jen kliknout na tlačítko „extrakce klausulí ze zadání“. Proces extrakce je spuštěn, vždy se vyextrahuje ze zadání jedna klausule a celý krok je vyhodnocen.

5.3.1 Rezolvování zaškrtlých řádků

Po extrakci klausulí se uživateli zpřístupní možnost provádět rezoluci. Rezoluce se opět provádí po krocích, v každém kroce uživatel musí zatrhnout právě dva řádky tabulky s klausulemi, zapsat do prázdného řádku klausuli, která podle něj bude výstupem operace rezoluce nad zaškrtlými klausulemi. Po zaškrtnutí řádků a zapsání formule může uživatel kliknout na tlačítko rezoluce dvou řádků, čímž spustí samotnou rezoluci.

5.3.2 Nápovědy v programu

První nápověda je uživateli k dispozici během převodu formule do klausulární formy. Nápovědu je možno spustit pro každý krok převodu tím, že se klikne na tlačítko „Nápověda“ (viz. obrázek 12). Druhá nápověda je uživateli k dispozici během procesu extrakce klausulí ze zadání. Nápovědu je možno zobrazit stisknutím tlačítka „nápověda k extrakci“ (viz. obrázek 13).

Třetí nápovědu může uživatel využít v procesu rezolvovaní klausulí. Nápověda se spustí při stisknutí tlačítka „nápověda k rezoluci“ (viz. obrázek 13).

5.3.3 Automatická rezoluce

Program uživateli dává možnost využít automatický proces extrakce klausulí a rezoluci klausulí až do nalezení/nenalezení sporu. Proces se spustí kliknutím na tlačítko „Strojová rezoluce“.

Obrázek 13: Grafické uživatelské rozhraní programu 2.

6 Závěr

V této práci je v ucelené formě sepsán souhrn teorie predikátové logiky prvního řádu. Dále jsou v této práci prezentovány v ucelené formě syntaktické důkazové metody v jazyce predikátové logiky prvního řádu (vyučovaných v předmětu matematická logika).

Byl vytvořen samostatný programový modul schopný vést uživatele krok po kroku procesem obecného dokazování platnosti formule za pomoci obecné rezoluční metody. Programový modul pracuje efektivně, je schopen napovídat, hodnotit a kontrolovat kroky uživatele. Program se jednoduše a intuitivně ovládá. Pro ukázku funkčnosti vytvořeného programového modulu byl vytvořen také demonstrační web.

Zdrojové texty k programu byly řádně okomentovány a byla také vytvořena programová dokumentace.

Další vývoj programového modulu je možný. Nejužitečnější by bylo rozšířit modul o vedení statistik znázorňujících to, jak se uživatel v problematice obecné rezoluční metody zlepšuje v závislosti na počtu vyřešených příkladů nebo v závislosti na čase. Další možnou oblastí, kde by se mohl modul rozšířit je automatizované generování zadání. V současné době program pouze vybírá zadání z množiny všech zadání. S rozšířením automatického generování by vymizela nutnost zadávat množinu možných zadání.

7 Reference

[1] doc. RNDr. Marie Duží, CSc., DUŽÍ, Marie. *Matematická logika*. Ostrava, 2003. 131s.

[2] Doxygen, *Stack.nl*[online]. 2011 [cit. 2011-05-04]. Doxygen. Dostupné z WWW: <http://www.stack.nl/~dimitri/doxygen>.

[3] Matematická logika, *Cs.vsb.cz*[online]. 2011 [cit. 2011-05-04]. Matematická logika. Dostupné z WWW: <http://www.cs.vsb.cz/duzi/PresentaceMat-Logika7-06.ppt>.